Simulating Escrow Transactions

Robert Crowell

MIT
crowell@mit.edu

Lyric Doshi

MIT
lyric@mit.edu

Abstract

Long running transactions, such as online purchases, severely reduce throughput under standard locking protocols such as two-phase locking. We investigated the use of escrow transactions to remove the need to wait for locks, and thus reduced the average transaction runtime. Escrow transactions allow transactions to borrow a quantity from the record until the transaction aborts or commits and to update the record appropriately. By escrowing some quantity away from the record, the record itself remains available for other transactions to read and escrow from as well. Patrick O'Neil's paper suggested this approach would theoretically reduce transaction run time, so we built a simulator to test the idea. The simulator models transactions under various workloads interacting with locks, a disk manager, and an escrow journal to compare the performance of escrow transactions against reads and writes using two-phase locking.

We found that escrow transactions significantly outperformed the two-phase-locking protocol. The locking scheme often spent nearly half its time waiting for locks, while the escrows were only blocked by the availability of the disk. When lock contention was high and transactions were likely to deadlock when using two-phase-locking, escrow transactions average run time was nearly 25 times faster than two-phase locking. Future work will involve extending the simulator to model a distributed database over a wide area network. We will investigate using escrow transactions and data redistribution schemes to reduce network communications and hence cut down the overall transaction run time.

1 Introduction

To motivate this paper, we consider a situation with long running transactions on a database that holds records that are incremented and decremented. Under standard two-phase locking, the database would perform very poorly under high load since only transaction could access and use a record at any given time. For example, consider an Amazon.com buyer who adds some books to his/her shopping cart and then spends several minutes looking for his/her credit card and confirming his/her purchase with friends on the phone before completing the transaction. We do not want to block others from purchasing this book until the first user's transaction completed, but also want to guarantee this user the inventory he/she placed in the shopping cart.

To address this problem, we investigated Escrow transactions. At a high level, an Escrow transaction borrows some quantity from the record and sets it aside for later use. Thus, the quantity is allocated for the transaction, but the record itself is not locked. As a result, other transactions may also escrow portions of the total quantity for themselves. If a transaction commits, the escrowed quantity is removed from the original record. If a transaction aborts, the escrowed quantity is returned to the record and available to be escrowed by future transactions.

We built a simulator to emulate a local database system using standard two-phase locking and escrow transactions. The code models transactions using locking, updating an escrow journal, and interacting with the disk. We then ran the simulator on various workload scenarios to investigate how much better escrow transactions performed compared to two-phase locking. Specifically, we profiled how long transactions take to complete, including how long the spend waiting for locks, waiting for the disk, and using the disk.

Once the this local database is built and tested, we hope to expand the idea of using escrow to speed up transactions between a user and the database to be used with a distributed database over a wide area network. We envision a scenario where data is partitioned among sites in a distributed database where each site then serves local queries. When a site runs out of inventory, it can attempt to escrow additional inventory from other sites. In a distributed database, the network latency becomes a major cost factor, so reducing the amount of communication between sites should help improve performance and average transaction runtime. Developing a simulator for this set-up and evaluating its performance will be the subject of future work to extend the project presented here.

The remainder of the paper is organized as follows. In Section 2, we discuss previous work with escrow transactions. Section 3 explains our general approach in setting up the simulator. In Section 4, we explain our testing and verification methods. In Section 5, we present and analyze our simulation results. Finally, in Section 6 we discuss ideas for future work.

2 Previous Work

The Escrow Transaction was first proposed by Patrick O'Neil in 1986 in *The Escrow Transactional Model* [4]. O'Neil describes a transaction that can escrow away a certain amount from a record for later use by the transaction. The purpose of the escrow is to avoid locking any record for any extended period of time. The record is only locked long enough to decrement/increment a certain amount and update an escrow journal described below. Thus, the escrow allows many transactions to interact with a single record concurrently and efficiently and also avoids deadlock situations. Escrows are not a blocking operation: they are either granted or denied if the quantity is not sufficient.

When performing an escrow, the transaction requests a certain amount from the record and also submits a test that must be satisfied. For example, in the case of inventory, a transaction may request n units so long as the inventory does not fall below zero as a result. An escrow journal is maintained for each record to track outstanding escrows. The journal maintains three fields: inf, val, and sup. The inf field represents the lowest value the record may achieve, i.e. the value if all decrements are committed and all increments are aborted. The sup field represents the opposite: the value if all increments are committed and all decrements are aborted. Finally, the val field represents the value of the record if all outstanding transactions commit. As each transaction commits or aborts, these fields are updated accordingly.

Whenever an escrow is requested, its test is applied against the inf field for a decrement and against the sup field for an increment. In other words, considering the worst case of the lowest or highest value the record could attain, we still want to make sure the escrow test is satisfied. An escrow request must also not violate any of the tests of currently outstanding escrows. That is, if a previous uncommitted/unaborted escrow required that the field not fall below 40, then no future escrow may push the field below 40 until that first escrow commits/aborts.

In O'Neil's paper, he describes escrowing a certain amount and then using it. After a transaction requests a certain allocation, it may choose to use only part of it based on further decisions made during the processing of the transaction. At commit, the used quantity in a decrement transaction will remain decremented from the inf and val. The sup will be reduced accordingly since now the greatest value the record can achieve is reduced by the committed amount. Any quantity that was escrowed, but unused will be returned the record and then inf and val will be increased accordingly. On abort, the entire escrowed amount is returned to the record and only inf and val need be updated. The operations are parallel for increment transactions, except sup and inf are reversed. Given the theory from O'Neil's paper, we decided to implement a simulation to test the theory and empirically evaluate the performance gain.

3 General Approach

In order to evaluate the performance of two phase locking and escrow operations, we built a database simulator in Java capable of processing hundreds of simultaneous transactions. While the simulator itself is not multi-threaded, it is capable of modeling a multi-threaded database system with multiple processors. External resources used by transactions, such as the current system time and the disk, are simulated in order to improve running times. When comparing locking to escrow transactions, we favored heavily strategies which resulted in the shortest average running times for a transaction. However, we also profiled each individual transaction in the simulator, obtaining detailed information about how much time the transaction spends obtaining locks, waiting for disks to become free, and reading and writing from the disk. While seeking the lowest average transaction running time allows us to prefer one strategy over another, detailed transaction profiles allow us to see how transactions spend their time and may, in the future, allow us to improve running times further.

When designing the simulator we considered two separate time domains: simulated time and real time. Simulated time is governed by the simulator itself; an internal clock is maintained that is advanced as transactions make progress. On the other hand, real time is governed by the speed of the particular machine running the simulation, as well as the efficiency of the code, the Java virtual machine, and a host of other factors. While we wanted the simulator to run as quickly as possible so that we could evaluate the performance of heavy workloads with large amounts of contention, we also wanted the results of the simulation to be the same regardless of the particular hardware running the simulator. Therefore, the simulator maintains its own notion of time that is independent of the actual system clock, and the performance of transactions is based on the simulator's internal clock.

The simulation is designed to provide a first approximation to the performance of an actual database system. After all, our goal is to evaluate algorithms and policies rather than to build a working database. In accordance with our objectives, the simulator makes approximations and simplifications where appropriate. For example, the CPU costs of performing logic within a transaction (performing a comparison or multiplication, for example) are assumed to be small and are not considered when computing a transaction's running time.

Furthermore, our simulated database system does not make use of a buffer pool. While the decision to exclude a buffer pool was partially motivated by time constraints, we also feel that this simplification is a fairly realistic one. A typical web service such as amazon.com will respond to queries involving a large percentage of the database at any given time as users make a wide diversity of purchases, and it is likely that a buffer pool using standard replacement strategies would have low hit rates. Thus, every record a transaction reads requires a disk action, and every update is immediately written to disk on commit. Furthermore, a single escrow operation requires four disk accesses; an initial read and write to read and update the escrow journal, and a second read and write to update the escrow journal to commit the escrow operation. As can be seen in Figure 2, the lack of a buffer pool can lead to poor escrow performance in low contention situations.

Even in the absence of real time performance considerations, it is valuable for the simulator to have precise control over the scheduling of processes within the simulated system. Thus, a single thread is used in the simulator, allowing our Java code to implement any scheduling algorithm desired by determining how transactions are allowed to do work and make progress. In the system our simulator models, we assume that the machine has enough processors to dedicate a single CPU to each currently-running transaction. Therefore, each transaction is run in a round-robin fashion during simulation and is allowed to perform one "unit of work" for each tick of the simulator's internal clock. The lock manager is allowed to process lock requests from its queue, the disk is allowed to perform part of a read or write, and transactions are allowed to request locks and schedule disk operations. This architecture allows us to approximate a multi-threaded system and makes the simulator's results independent from the machine it is actually running on.

The simulation is made up of two types of objects: scheduler objects and resource objects. Scheduler objects are responsible for enqueuing disk requests, for obtaining and releasing locks, and for deciding when transactions should be aborted or committed. Resource objects, on the other hand, do the actual work of the database. The disk manager, for example, maintains a request queue that is populated by transactions and performs the simulated disk reads and writes as requests are popped off the queue. Both scheduler objects and resources have a notion of performing one unit of work; scheduler objects check the status of their outstanding requests and schedule new actions as appropriate, while resource objects modify the resources of the system. A resource request may require multiple units of work (this, multiple ticks of the simulation clock) to complete, even after it has been popped off the queue. To simulate realistic disk reads and writes, for example, a single read or write requires 5 units of work to complete if a unit of work represents 1 ms. These values are based on a Western Digital RAPTOR hard disk drive, but can be easily changed to reflect a different hard drive's specs.

3.1 Simulator

The simulator is the top-level object in a database simulation. While the simulator is capable of supporting multiple database sites connected over a network, the workloads we tested involved only a single database site that contained the entire dataset. The simulator maintains the simulation time clock. After the clock ticks, the simulator allows the network to do one unit of work. While the network module isn't yet implemented, it will be responsible for forwarding packets between sites when modeling a distributed database system. After the network has done a unit of work, every site is allowed to perform a single unit of work, one after another. Once the network and all the sites have done a unit, the simulation clock ticks and the network and all sites are allowed to work again. This design allows the simulator to simulate concurrently running sites, since each is allowed to do a single unit of work within the same tick of the simulation clock.

3.2 Workloads

Within the simulator, we create workloads of transactions for each site to run. Each workload is associated with a single site and consists of a number of transactions and a start time for each one. Whenever the simulation clock reaches the start time of a transaction, the workload enqueues that transaction to its site. We set up our current system to schedule n transactions to each have uniform probability of starting some time within a given range of start times, but this can be adjusted as desired.

3.3 Site

Like the Simulator object, a site is responsible for scheduling work rather than for performing work itself. A site has a private set of transactions, as well as a unique disk manager and lock manager. Each time the site is allowed to perform a unit of work, it allows its lock manager, disk manager, and escrow manager to perform a unit of work. Once each of these resources has done its unit of work, the site allows each of its currently active transactions to do a unit of work. Afterwards, the site removes all complete transactions from its internal list and yields back to the simulator object, which then allows the next site to run.

3.4 Transaction

The simulator does not support SQL or any standard query language. Designed to test the performance of transactions rather than to act as a real database system, we decided it was more important to accurately model the behavior of a database than to make it easy to write a transaction. After all, a small library of transactions can be used to model a wide variety of workloads simply by varying the number of individual records involved and the number of concurrent transactions. Thus, a transaction consists of a block of Java code that performs some set of database actions when executed.

Transactions are allowed to perform only one action at a given time, and all actions in a transaction must happen serially in the order specified in the transaction's Java code. When a transaction is allowed to do a unit of work, it checks whether or not its most recent outstanding resource request (disk read or write, lock request, etc.) has completed. If not, the transaction yields back to its site, allowing the next transaction to perform a unit of work. On the next tick of the simulation clock, the transaction will again check the status of the outstanding request. If the request has been granted, the transaction immediately schedules its next request and yields to the site once again. The CPU cost of executing the logic contained within the transaction is not accounted for. (If a transaction wants subtract 1 from a record's value, for example, it finishes the read and schedules the write in the same unit of work. The amount of time required to perform the actual subtraction is assumed to be small, and therefore requires 0 units of work to complete.)

Transactions may be commit, abort, or restart. When a transaction commits it writes back all of its changes to disk and then exits. Transactions which use two-phase locking write to disk only when they commit, simplifying the process of aborting or restarting a transaction. Because a transaction writes to disk only while committing, a locking transaction never needs to write to disk during an abort or restart. Escrow transactions, however, need to write to disk to remove themselves from the escrow journal whether they commit or not. When a transaction aborts or restarts, its outstanding resource requests are not removed from the disk request queues. Locking transactions only perform disk reads until they commit; leaving disk read requests in the queue occupies the disk unnecessarily but does not result in inconsistencies. Escrow transactions write to disk before they commit; however, escrow transactions enqueue an additional disk write when they abort or restart to update the escrow journal. Thus, leaving the outstanding requests in the disk queue is not only correct but also essential for correct results in this scheme.

3.5 Lock Manager

Every individual database site contains its own local lock manager. The lock manager maintains a unique lock object for every record that is currently in use by some transaction on its site. The lock object records a list of owners for the record, whether it is owned SHARED or EXCLUSIVE, and an ordered list of requests for the lock which haven't been granted. When the lock manager performs a unit of work, it allows each of its locks to service as many requests off their individual queues as possible. Once a transaction is popped off a lock's request queue and can be granted the lock, no additional units of work are required before the transaction may begin using the record it has obtained a lock for. Thus, a transaction may begin using a record after only one round of waiting if the lock is unowned and no higher-priority transactions are in its queue.

The lock manager chosen for our simulated workloads uses the wound/wait deadlock prevention algorithm. This algorithm assigns every transaction an id based upon when it was started which acts as its priority. Earlier transactions have higher priority and may restart any transactions with lower priority (started later) whenever they want to acquire a lock that is currently unavailable. Thus, the lock request queues for each lock maintained by the lock manager are ordered by transaction priority, with the highest priority transactions at the head of the queue. The wound/wait algorithm ensures that deadlocks are corrected as soon as they might arise, making it unnecessary to implement any further deadlock detection or correction schemes.

3.6 Disk Manager

The disk manager controls read and writes to stable storage. It may control one or having multiple disks, with data distributed by key among however many disks are initialized. Reads/writes are enqueued to the appropriate disk in the order they are received. The disk manager is backed with an in-memory hash table to hold its actual data, but uses counters to simulate the time required for a disk read/write operation. Each read and write takes a certain amount of simulation time, based on the seek time and read/write time for a real disk. When the disk manager does a unit of work, it effectively decrements a timer representing the time required for the current read/write. For example, if a read takes 10 ms and the current time steps are .1 ms, then a read will last for 100 time steps. When the counter reaches zero, the read/write is completed and then next one begins. At this time, the written value is stored to the hash table and the read data is made available to the transaction that made the read request. The disk manager also updates the transaction's state as the transaction waits to use a disk and then actually gets to read/write it so that we can profile the time a transaction spends waiting for and using the disk.

3.7 Escrow Journal

The escrow journal implements the ideas presented by O'Neil to make escrow transactions possible. It maintains an entry for each key that holds outstanding transactions on the key and tracks how much has been escrowed by each. The journal is also responsible for making sure all escrow tests are satisfied and maintains the inf, val, sup for the key. On commit and abort, all keys touched by a transaction are updated appropriately. To emulate a transaction grabbing a lock, decrementing (incrementing) a record, and writing it back, we allow the escrow journal to make a special request to the disk manager that involves effectively an atomic read-decrement-write (read-increment-write) operation. Just like any other disk manager request, this escrow request to the disk manager must queue until the disk it wishes to modify is free.

Since we do not have a buffer pool, we assume the escrow journal must read and write to disk every time an escrow is requested. It must also read and write to disk for each key when the escrow journal is modified and during aborts/commits. For simplicity, we ignore the use syntax presented by O'Neil and assume that a transaction will always want to use everything it escrows. Just as with the disk manager, the requesting transaction must wait for the escrow journal to do units of work to access the disk manager and update its records before it can learn whether the escrow succeeded.

Currently we support two different escrow semantics. The first is a full escrow, where the escrowed amount is either as requested or zero if the escrow request cannot be fulfilled. The second is an escrow-up-to, in which the escrowed amount is either as requested or as much as could be escrowed without violating any tests. For example, if the inf is 5 and the test is greater than 0, then a full escrow would return 0 for an escrow request of 10. However, an escrow-up-to request would return 5.

4 Testing and Verification

It is notoriously difficult to test and debug database systems. It may be even harder, in fact, to test simulators of database systems because nothing is done with the results read and written by the transactions running in the simulator. Our simulator was designed to run a transaction to completion and to immediately kill a transaction when it completes. However, in order to verify that the simulation was working correctly, we modified the transactions to optionally return a results object indicating the values they read, wrote, or escrowed. This functionality was utilized in the almost three-thousand lines of JUnit tests we created, in which we ran one or two transactions simultaneously or back to back.

While the test transactions were running, we carefully monitored their status as reported in real-time by the transaction profiler to ensure that each resource request took exactly the right amount of time and that the transactions progressed in the right order at the right times. After the transactions completed we then tested that the transactions either committed or aborted successfully and that the values they read or modified were correct. Because the tests were set up to run only one or two transactions simultaneously, such exact monitoring was possible as we knew exactly when and for how long transactions should be operating. As the number of concurrent transactions increases, it becomes harder to compute running times by hand and makes sense to run a simulation instead. Because we wrote unit tests to validate correct behavior for the resources, such as the lock manager and the disk manager, we believe these resources work correctly. And, since we also wrote unit tests which verify that a site allows its resources and its transactions to perform their units of work in the correct order, and since our unit tests show that a site's transactions generate correct results in the appropriate amount of time, we feel that the transactions we wrote and the sites that execute them operate correctly as well.

5 Performance of Simulated Workloads

We simulated various workloads to evaluate the performance of locking and escrow transactions. To make a fair comparison, we restricted both workloads to simple decrement requests. To decrement a record, locking transactions acquire a shared lock on that record, read the record, acquire an exclusive lock on the record, and then write the record back during commit. The escrow transactions escrow and immediately use a certain quantity from the record. While the simulator is capable of scheduling transaction start times in an arbitrarily complex way for a workload, each of our tests started all transactions within the same two millisecond window in order to increase the amount of resource contention and deadlock. While we performed simulations that varied the delays, number of concurrent transactions, probability of deadlock, and the number of available disks to generate 88 different tests, we present a representative sample of our results here. In almost every test, escrow transactions outperform locking transactions significantly.

The first set of tests involved read-write and escrow transactions only. As their names suggest, these transactions either read and write to a single record using two-phase locking, or escrow some quantity from a single record. In all of the workloads presented in Figure 1, all of the transactions attempt to decrement the same record, resulting in lock contention and frequent deadlocking. The wound/die deadlock prevention algorithm results in many restarted transactions, which must redo all of their lock acquisitions and disk reads on every restart. As shown in Figure 1, escrow transactions have a much lower average runtime as compared to two-phase locking. As the number of simultaneous transactions increases, two phase locking has poorer and poorer performance due to increased lock contention and numerous restarts resulting in a many repeated disk accesses. When the workloads are modified such that each transaction attempts to decrement a unique record in the database, there is no lock contention and almost all of a locking transaction's time is spent waiting for and using the disk. As can be seen in Figure 2, two-phase locking actually outperforms escrow transactions in this case. This is mostly due to the fact that a single escrow transaction requires four disk accesses (a read and write to update the escrow journal, and a second read and write to commit the escrow) whereas locking transactions only require two. However, the addition of a buffer pool would likely improve the performance of escrow transactions such that the average running times in both cases would be almost identical.

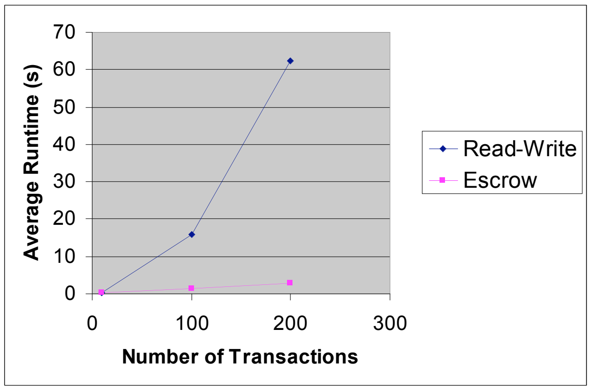

Figure 1: Performance of read-write (locking) vs escrow transactions modifying a single record.

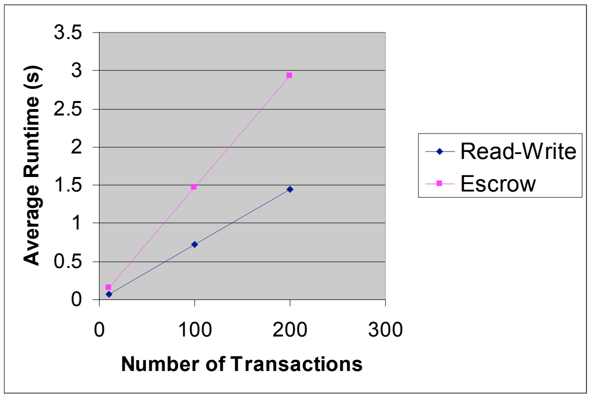

Figure 2: Performance of read-write (locking) vs escrow transactions each modifying a unique record.

As shown in Figures 3 and 4, profiling the read-write and escrow transactions that all modify the same record reveals that escrow transactions are bound by the disk queue; regardless of the number of keys the escrow transaction workloads modify, they have no need to wait for locks and never need to be restarted due to deadlocking. The read-write transactions that all access the same record, however, spend a significant amount of time waiting for locks. As shown in Figure 3, read-write transactions also spend a proportionally large amount of time reading and writing to disk because they must read the shared record again each time they are restarted. When the second disk is added, however, read-write transactions and escrow transactions both spend the majority of their running time waiting for in disk queues, as seen in Figures 5 and 6. This is due to the fact that locking transactions no longer wait for locks and never need to be restarted due to deadlock.

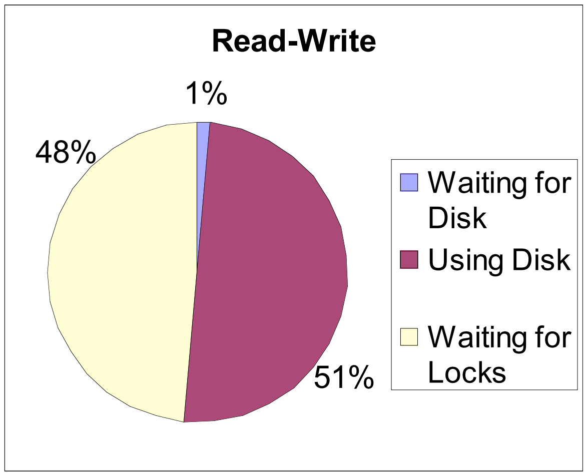

Figure 3: Profile of a typical read-write (locking) transaction modifying a single heavily-used record.

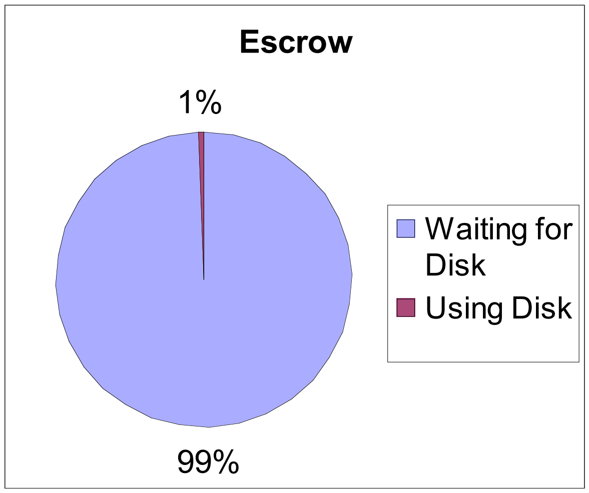

Figure 4: Profile of a typical escrow transaction modifying a single heavily-used record.

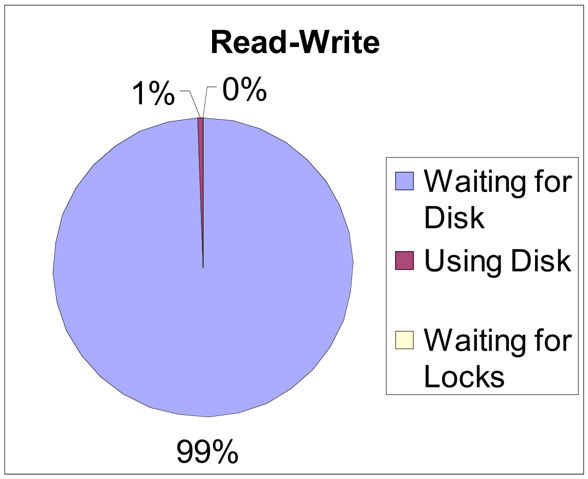

Figure 5: Profile of a typical read-write (locking) transaction modifying a single record that no other transaction is accessing.

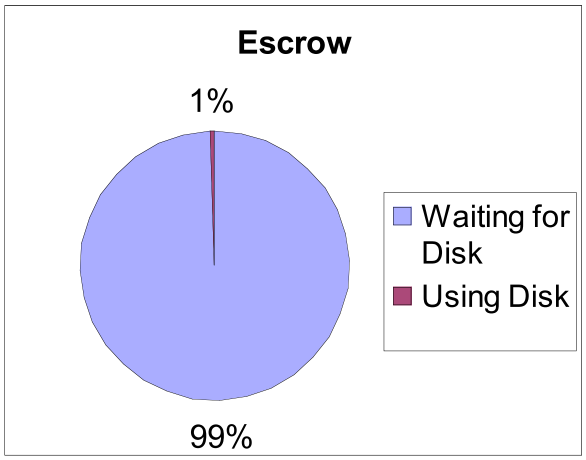

Figure 6: Profile of a typical escrow transaction modifying a single record that no other transaction is accessing.

The second set of tests involved transactions that perform read-write and escrow operations on two different records A and B. While this decision should have little impact on the performance of escrow transactions, it makes deadlock more likely when using two phase locking. To increase the number of transaction restarts that occur in the locking case, the records transactions modify were intentionally chosen to cause deadlock by having half of the transactions lock A before B, and the other half acquire B before A.

The performance of the workloads presented in Figures 7 and 8 illustrates the dramatic difference between locking and escrow transactions in this worst-case scenario for locking. All transactions were designed to decrement the same two records in order to increase the chance of deadlock. Figures 7 and 8 illustrate the performance of workloads that vary the number of simultaneous transactions between 10 and 200. The same workloads were run on a simulated system with one and two disks (adding more disks would not improve performance, since all transactions modify the same two records). As can be seen in Figure 7, escrow transactions perform much better than locking transactions for the same set of decrement requests. As expected, the performance of locking decreases dramatically as the number of simultaneous transactions increases, while escrow transactions see little performance penalty. Increasing the number of disks improves the performance of both locking and escrow transactions, as shown in Figure 8; because the two records used by the transactions are placed on different disks, the disk queues are half as long and operations for two different transactions can be performed simultaneously. Because there are no buffer pools in the simulation, performance increases dramatically with an additional disk.

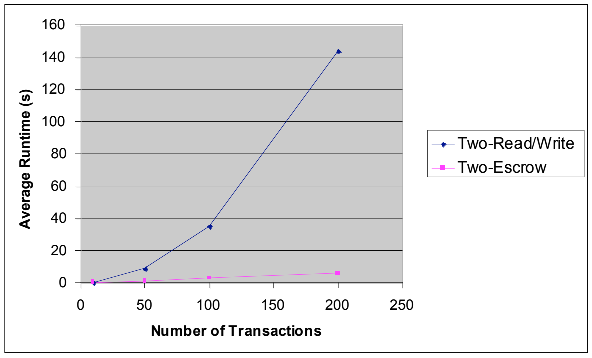

Figure 7: Performance of two-read-write (locking) vs two-escrow transactions with both records on the same disk.

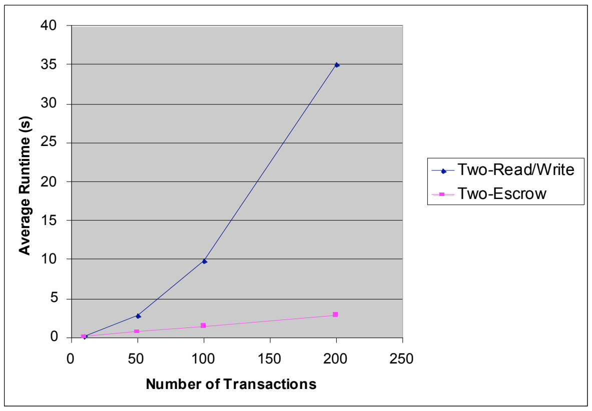

Figure 8: Performance of two-read-write (locking) vs two-escrow transactions with each record on its own disk.

As can be seen in the transaction profiles presented in Figure 9, the two-read-write transactions spend about a quarter of their running time waiting to acquire locks. They also spend a lot of time reading disk; the lack of a buffer pool means that a transaction must redo its writes every time it is restarted by the wound/die deadlock prevention algorithm. Thus, a transaction that is restarted just before it commits must read the two records again when it restarts. As shown in Figure 10, the two-escrow transaction spends almost all of its time waiting for disk, because the IDE request queue is their only bottleneck, and because escrow transactions never need to restart or abort due to deadlock. Even with the addition of a second disk, Figure 12 shows that escrow transactions still spend 99% of their time waiting for disk. The two-read-write transaction, on the other hand, spends proportionally more time waiting for locks that before, as can be seen in Figure 11. This is due to the fact that adding a second disk reduces the time transactions spend waiting for disk but does not alter the locking behavior at all.

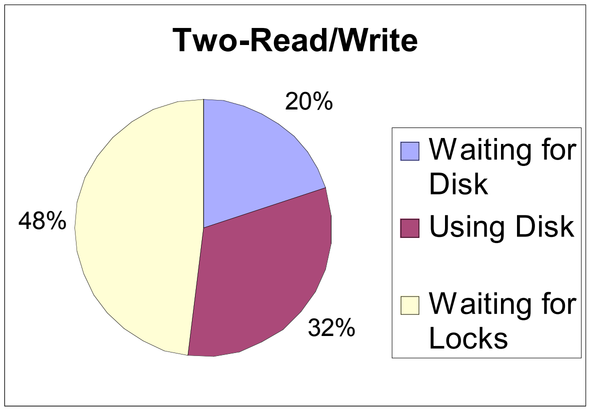

Figure 9: Profile of a two-read-write (locking) transaction modifying two heavily-used records on the same disk.

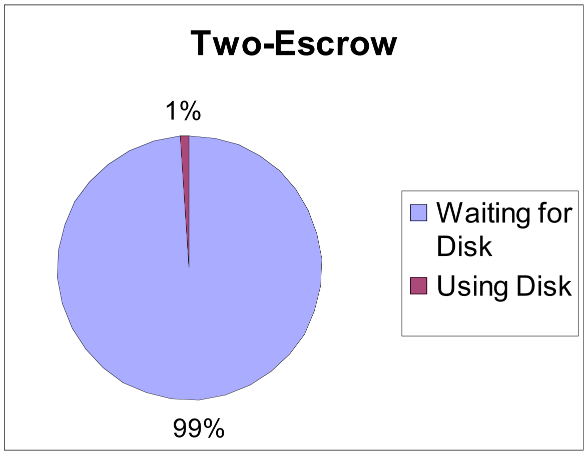

Figure 10: Profile of a two-escrow transaction modifying two heavily-used records on the same disk.

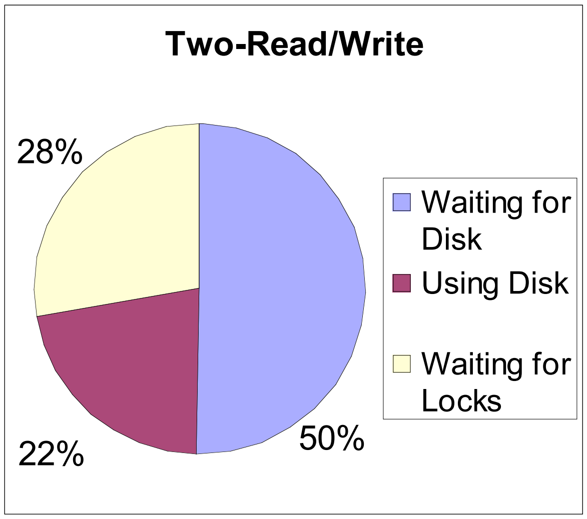

Figure 11: Profile of a typical two-read-write (locking) transaction modifying two heavily-used records each on different disks.

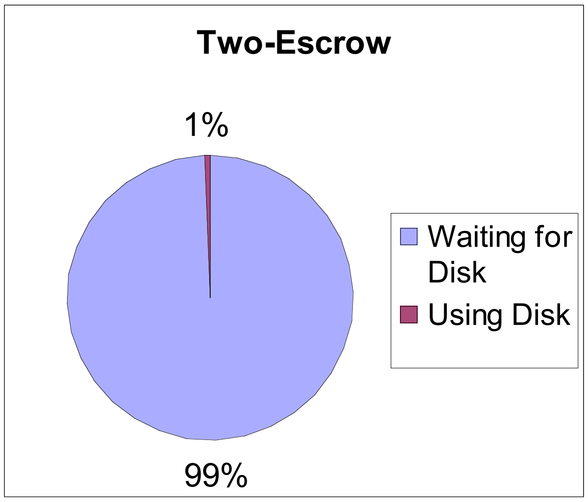

Figure 12: Profile of a typical two-escrow transaction modifying two heavily-used records each on different disks.

6 Conclusion and Future Work

We have presented a simulator for a local database that takes into consideration locking, disk management, and escrows. We found that as the possibility for lock contention increases, escrows outperformed standard two-phase locking schemes. Furthermore, we found that escrows performed much better than two-phase locking when deadlocks were made very likely. Because escrows do not block each other from accessing a record, escrows primarily spent time waiting for the disk and writing results and journal entries to disk. Meanwhile, the locking protocol spent nearly half its time waiting for locks. In situations of high contention, escrows allow for higher transaction throughput because they do hold locks on records until their transaction completes. In low contention situations, the performance between the two techniques becomes comparable because time is not wasted waiting for locks and restarting transactions.

Now that we have shown that escrows make a significant performance improvement in local transactions, we wish to expand our simulator to support a distributed database over a wide area network. A distributed database must also consider the latency of transferring data between sites, which can become a significant fraction of the transaction running time when distances are large and network communications are frequent. For example, a central database that all sites must communicate to will likely incur many long network round trips and also bottleneck all requests at one site. Data replication results in frequent communication to keep sites in sync because our data set is constantly changing with decrements and increments. Thus, we wish to investigate partitioning the quantity among all the local sites so that most queries can be resolved by the local site. Only when the local site is unable to satisfy a query will it contact other sites to obtain more quantity. To make these queries to other sites efficient, escrow transaction can be used so that sites don't block on remote accesses to their records. Kumar and Stonebraker as well as Krishnakumar and Jain have presented ideas relating to managing a distributed database with partitioned data and escrows between sites [2, 3]. Both papers agree that in theory, this set up will help reduce communication between sites and keep transactions from blocking each other. In 2001, Cetintemel et al actually created a simulation to test escrow between remote sites [1]. They also tested various redistribution schemes. For example, when a site fell short of supply, one scheme required it to randomly query other sites until its needs were fulfilled. In another, the sites constantly update each other on their current inventory so that requests can be made strategically to remote sites that are likely to fulfill the site's needs. Finally, they tested a scheme the tracked the demand at each site and proactively redistributed quantity such that sites with greater demand were given greater quantity.

In our own implementation, we wish to test out similar redistribution methods as well as try out some of our own. We hope that our lightweight simulator will also allow us to investigate a wider range of parameters. For example, Cetintemel et al only tested having up to ten sites in their distributed network, but we hope to test a hundred or even a thousand. We may also compare performance against two-phase commit or some locking protocol as a baseline to see how escrows perform, but the more interesting part of this investigation will be seeing what redistribution and learning schemes we can apply to properly partition data such that network communications are significantly reduced. We will work on this project as a UAP under Samuel Madden.

References

[1] U. Cetintemel, B. Ozden, M. Franklin, A. Silberschatz. Design and evaluation of redistribution strategies for wide-area commodity distribution. *Proceedings of the 21st International Conference on Distributed Computing Systems*: 154-164, 2001.

[2] N. Krishnakumar and R. Jain. Escrow techniques for mobile sales and inventory applications. *Wireless Networks*, 3(3): 235-246, 1997.

[3] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. *ACM SIGMOD,* 17(3): 117-125, 1988.

[4] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4): 405-430, 1986.