Constrained Snapshot Isolation: Leveraging Transactional Logic to Solve the Write-Skew Problem

Jeremy Liu
Yale University
jeremy.liu@yale.edu

David Marcano
Yale University
david.marcano@yale.edu

Kshitijh Meelu
Yale University
kshitijh.meelu@yale.edu

Krishnan Srinivasan
Yale University
krishnan.srinivasan@yale.edu

Abstract

Snapshot isolation is a commonly used isolation level in most MVCC database systems today. Write-skew is a well-known flaw of snapshot isolation which makes snapshot isolation unserializable. While other attempts to fix this issue result in making snapshot isolation serializable, we present a new isolation level, known as constrained snapshot isolation, that solves write-skew by only aborting unserializable workloads that will break a transaction's control flow logic. This approach, when applied to a modified Smallbank benchmark, shows performance improvements over serializable isolation methods, while only showing slightly less throughput than regular snapshot isolation. Thus, constrained snapshot isolation is a new isolation level with higher throughput than serializable solutions to the write-skew problem.

1. Introduction

In all database management systems (DBMS), it is crucial to select an appropriate concurrency control mechanisms for the intended usage pattern(s) of the database. Two phase locking, optimistic concurrency control (OCC) and serializable multi-version concurrency control (MVCC) are three common concurrency control schemes that maintain the serializability of transactions. The principle of serializability allows application programmers to use the database as if transactions were executing one after the other and to view a transaction as if it is the only transaction operating on the database at a time.

However, a key issue with serializable transactions is that they often slow down performance due to transaction blocking or frequent aborts. For example, read-heavy transactions (e.g. retrieving bank account balances) do not require such strict concurrency limitations and should be allowed to execute concurrently. Snapshot isolation (SI) is a weaker isolation level that addresses this problem. On a basic level, SI allows each transaction to execute on a previous "snapshot" of the database. This ensures that read operations never block and only write-write conflicts need to be resolved. SI is commonly implemented in MVCC as the multi-versioning allows for easy tracking of snapshots with a verification stage to abort write-write conflicts. SI tends to work well for read heavy transaction workloads that take advantage of its optimistic assumption.

Although it provides greater concurrency, SI is not a serializable isolation level because of the write-skew problem. Generally, write-skew occurs when two read-write transactions read and then independently update companion records on the database. Operating on its own snapshot, each transaction believes it is committing a change that upholds the constraints (e.g. the sum of a customer's checking and savings accounts must be greater than 0) of the transaction. But when executed concurrently the transactions' changes could violate a critical constraint of the system.

To solve the write-skew problem, Cahill et. al. [5] proposed serializable snapshot isolation (SSI), a version of SI that keeps track of each transaction's dependencies. This method detects when transactions are concurrently reading/updating the records with cyclic dependencies, and aborts them when found. SSI conforms to a serializable isolation level that misses some of the benefits of SI, as it will abort all un-serializable workloads, including those that do not break any constraints at the transaction or database level. As such, these transactions could have been executed in any order without any change to the database. Hence, this motivates a concurrency scheme and isolation level that only abort prob-

lematic write-skew transactions.

In the following sections, we describe constrained snapshot isolation (CSI), a new isolation level that we implement using a variant of Larson et. al.'s [9] MVCC mechanism to more accurately solve the write-skew problem of SI with faster performance than serializable isolation. The primary contribution is the introduction of a validation phase where the transaction collects fresh reads of the required values in its read set and then checks if the transaction truly violates the transaction constraint. If the constraint is violated, the transaction aborts. This approach is viable under our modified MVCC mechanism which makes fresh writes visible to concurrent transactions during constraint checking. This leads to only the problematic write-skew transactions being aborted.

The remainder of the paper is structured as follows: Section 2 provides an in depth exploration of SI, serializable isolation, and SSI. Section 3 contains an in-depth description of the CSI algorithm. Section 4 outlines our system design. Section 5 describes the experimental setup and results of running CSI, SI, and serial MVCC on write-skew transactions. Section 6 tabulates related work to the project. Section 7 outlines future work on the CSI algorithm. Section 8 concludes the report.

2. Motivation

We now discuss two standard isolation levels, snapshot isolation and serializability, along with a brief description of an example concurrency control scheme that implements each of the given isolation levels.

2.1. Snapshot Isolation

Snapshot isolation is a commonly used isolation level in systems with multi-version concurrency control schemes to support non-blocking reads. It works by assigning each transaction a unique begin timestamp (usually acquired from a global counter) and allowing that transaction to read any version with a timestamp lower than its begin timestamp. This ensures each transaction executes on a version of the database visible at the start of its execution. At the end of its execution, the transaction and any versions it writes to the database acquire an end timestamp, after which the transaction processor checks if the keys read by the transaction have gotten new versions that overlap the start/end timestamp interval of the current transaction. Under the SI scheme, the transaction then aborts and restarts, since the scheme does not use locks to ensure that transactions always read the most up-to-date version. However, note that in this scheme, reads are non-blocking, which is why this isolation level offers a performance boost in read-heavy workloads.

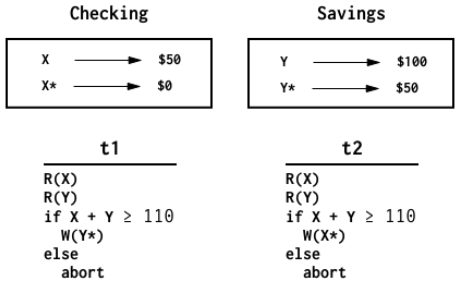


Figure 1: Example of the write-skew problem

It is well known that in exchange for performance advantages, lower levels of isolation such as SI are confronted with the issue of failing to serialize certain transaction workloads. This particular failure in the case of SI is referred to as the write-skew problem. This problem is illustrated by the following example, which is depicted in Figure 1.

Suppose we have two concurrent transactions, T1 and T2, each editing a different record, $\mathbf{X}$ and $\mathbf{Y}$. Both transactions will execute only if $\mathbf{X} + \mathbf{Y} \geq C$ (where C is a pre-defined constraint) and will abort otherwise. T1 and T2 must therefore read both $\mathbf{X}$ and $\mathbf{Y}$ and update the value to some new version ($\mathbf{X}*$, $\mathbf{Y}*$). Under SI, T1 and T2 are allowed to execute concurrently, and given the initial versions ($\mathbf{X}$ and $\mathbf{Y}$) that both transactions read, both T1 and T2 would commit successfully. However, this cannot be serialized into any order, as committing either T1 or T2 would cause the other transaction to abort. That is, committing T1 or T2 first would cause the other transaction to see either $\mathbf{X} + \mathbf{Y}* < C$ or $\mathbf{X}* + \mathbf{Y} < C$, respectively, leading to an abort.

The write-skew example breaks serializability as one of the two transactions should not have been allowed to go through. This issue, which is the main downfall of SI, has several solutions, the most popular of which is serializable snapshot isolation, a concurrency control scheme that implements the serializable isolation level.

2.2. Serializable Isolation

A stricter isolation level than snapshot isolation, serializable isolation simply looks to ensure the serializability of concurrently executing transactions [8]. That is, concurrent transaction execution must be done in such a way that the logic executed would be the same as if those transactions executed in some serial ordering.

There are many concurrency control schemes that guarantee serializability. One example of such is a lock-based timestamp-ordering MVCC scheme. In this system, a transaction reads from storage and runs its exe-

cution logic. Then, before applying its local writes to storage, the transaction ensures that none of its writes invalidate the reads of other transactions. If so, it commits; otherwise, it aborts and restarts execution. Note that this serializable concurrency control scheme may have blocking reads.

Another interesting concurrency control example, known as serializable snapshot isolation, combines serializable isolation with the non-blocking reads of SI and is explored below.

2.2.1. Serializable Snapshot Isolation

Proposed as a solution to SI's write-skew problem, serializable snapshot isolation [5] is a stricter concurrency control mechanism that enforces serializability in circumstances where write-skew may occur. Built on the theory presented by [2] and [7], SSI detects two consecutive *rw*-dependency edges involving two concurrently active transactions. In this context, a *rw*-dependency—also known as an anti-dependency—occurs when a transaction T1 performs a read on item $\mathbf{X}$ prior to transaction T2's write on that item. Write-skew involves at least two such dependencies forming a cycle.

To solve this, the system performs look-ahead operations within its read, write, and commit methods, and dynamically aborts transactions that may cause a cycle of anti-dependencies. Note that the system does not find cycles; instead, it detects whether a cycle may be present.

The benefits of this approach include system serializability and the upholding of SI's fundamental tenet (i.e. reads do not block writes). However, an important disadvantage is the presence of false positive aborts made by the system, in which case a transaction aborts unnecessarily. The decreased throughput due to aborts, as well as the increased checking of locks and flags, deems serializable snapshot isolation considerably slower than normal snapshot isolation. Thus, in attempts to serialize SI, much of snapshot isolation's initial benefits erode.

Part of the motivation for CSI arises from the need for a less strict solution to SI's write-skew problem than serialization. While this solution, presented by [5], fixes the serializability of snapshot isolation without the need to calculate expensive dependency graphs, it performs some additional locking logic and introduces additional aborts in attempts to ensure serializability.

3. Constrained Snapshot Isolation (CSI)

Our new isolation level is similar to snapshot isolation as explained above, but solves the write-skew problem without being serializable. It recognizes that serializable isolation levels abort or lock even in cases where anti-dependencies result in adequate outcomes. This paper's main contribution is the empirical proof that an isolation level closer to SI without write-skew can be achieved by guaranteeing that a transaction's control flow logic remains intact during execution. With the addition of a constraint set, a transaction can ensure that a set of constraints at the beginning of the transaction remains unchanged at the end. This is guaranteed to be better than serializable isolation because it avoids unnecessary aborts.

3.1. Example of CSI vs. Serializable Isolation

As mentioned above, most modern databases implement serializable isolation levels by keeping track of possible cyclic dependencies or locking. To reduce implementation costs and increase processing speed, these methods overestimate how many true cyclic dependencies appear between transactions, even when the results might be adequate. Suppose we have two concurrent transactions, T1 and T2 that both read records $\mathbf{X}$ and $\mathbf{Y}$, and T1 writes to $\mathbf{X}$ and T2 writes to $\mathbf{Y}$ as in Figure 1. If our system holds constraints on $\mathbf{X}$ and $\mathbf{Y}$, such that the sum of $\mathbf{X}$ in a CHECKING table and $\mathbf{Y}$ in a SAVINGS table must be greater than 0, then both T1 and T2 must check their reads against the constraint before committing. Assume they enter at nearly similar times so that, under CSI, they read the same values for $\mathbf{X}$ and $\mathbf{Y}$. Let $\mathbf{X}$ contain \$1 million and $\mathbf{Y}$ contain \$1 million, and T1 tries to withdraw \$100 from CHECKING and T2 tries to withdraw \$50 from SAVINGS. In SI, at constraint checking time, they will both pass and commit. In a serial isolation level, this dependency will be noticed and one of the transactions will abort or block. But, we see that since T1 and T2's constraints are not broken (even in combination after they both commit), their concurrent execution results in the same outcome as a serializable one. Specifically, this is the case from Figure 1 where the right hand side of the inequality is 0 instead and $\mathbf{X}$ and $\mathbf{Y}$ sum to a large positive amount. This means that allowing both of these transactions to act concurrently and then commit would not result in an inconsistent database state. In contrast, a serial isolation level will have a mechanism that either locks the tables, or performs an unnecessary abort of T1 or T2, thus butchering performance.

3.2. CSI's Constraint Path

In CSI, we create a data structure that builds a "constraint path" at the beginning of the transaction, and then checks against it at validation time. If the paths diverge at any point, then a constraint that was valid/invalid at the beginning of the transaction has now inverted, possibly due to a concurrent transaction's operations. This means that in the above example, we would construct the same path both at the beginning and at validation,

thus resulting in the allowance of both transactions to commit (which is the desirable outcome). In a situation where the constraint for T1 or T2 was met at the beginning but not at validation, it means that the other transaction must have decremented the value of CHECKING or SAVINGS so much so that the execution of T1 or T2 results in a negative CHECKING or SAVINGS (when C is 0). In this case of write-skew, CSI would abort one or both transactions depending on which constraints were broken.

4. System Design

The design of our system is inspired by the snapshot isolation MVCC scheme described in [4], with a modified storage system. Note that for the transactions described below, the read set, write set, and constraint set must be specified in advance. This is not a unique requirement, as many systems also require *a priori* information regarding the read and write sets of transactions. Additionally, the control flow logic must be stated in advance, which allows the construction of the constraint validation path.

4.1. System Overview (Transaction Processor)

Transactions that enter the system are handed over to a single thread that adds them to a shared-memory queue of transaction requests with their *status* as INCOMPLETE. The position of a transaction in this queue is initially dependent on when it was received by the thread—in some cases, as will be described, transactions may reenter this queue. This queue is read by the concurrently-running threads. A thread, when free, pops a transaction off the queue of requests and begins to handle it. Thus, each thread processes a subset of all transactions that enter the system and has equal access to all records.

Each transaction first changes its *status* to ACTIVE and obtains a unique, monotonically-increasing timestamp as its begin timestamp. Then the thread reads in the records in the transaction's read set and write set, as of the time indicated by its begin timestamp. More specifically, it stores pointers to the specific versions of those records from storage.

The thread then loops through the write set and initiates all writes. That is, it writes the given transaction's pointer into the storage metadata (as in [9]), specifically into the version the transaction initially read. This initialization fails in the case of a write-write conflict, which happens if another transaction is currently writing to, or has written to, the record the current transaction read. This step of initiating writes only succeeds if initializing writes to all records in the write set succeeds. Otherwise, the transaction aborts and reenters the queue of transaction requests (with its *status* as INCOMPLETE), allowing the thread to move to the next transaction in that same queue.

If the initialization of writes succeeds, the transaction's logic is executed. This includes all control flow decisions, reads from the stored pointers, and writes out to new local versions. Once this logic is run, the thread then populates these new local versions into storage, with the transaction pointer in the storage metadata (as in [9]). If the transaction aborted due to its logic, the thread empties the transaction's reads and writes, sets its *status* to ABORTED, and adds it to a queue of completed transactions.

Otherwise, the transaction obtains a second globally unique, monotonically-increasing timestamp as its end timestamp. This timestamp is used in the transaction's validation phase, at which point the thread reads the constraint set as of the time indicated by the end timestamp. As long as the constraint logic is validated and executes in the same way as it did in the transaction logic's initial execution, the transaction is said to ensure *constraint stability* and may change its *status* to COMMITTED. At this point, it replaces its pointers in storage metadata with its end timestamp and gets put on a queue of completed transactions.

Note that if it does not validate successfully, it goes through the same procedure as if its writes were not initiated. That is, the thread pushes the transaction back onto the queue of transaction requests with its *status* as INCOMPLETE.

In addition to understanding the way a thread handles each transaction, we must also discuss specific design decisions with respect to the storage handling.

4.2. Processing Read/Write-sets (Storage Manager)

During a transaction's execution, the handling of its reads and writes requires a data structure that, at the most basic level, has multi-version records. For the purpose of our CSI implementation, we used the C++ deque and map data structures for this part of the system. When a transaction is run, it has to determine what version of each record it will read, and this requires finding the version that is "visible" to the transaction. The visibility of a version is decided by three different timestamps: the version's begin and end timestamps, and the transaction's begin timestamp. In addition, there are three main cases when deciding whether a record is "visible" or not that have to be handled within the storage implementation.

When processing the read and write sets and deciding version visibility, one possible scenario is when the current transaction (let us call this T1) has a begin timestamp that is between a version's begin and end times-

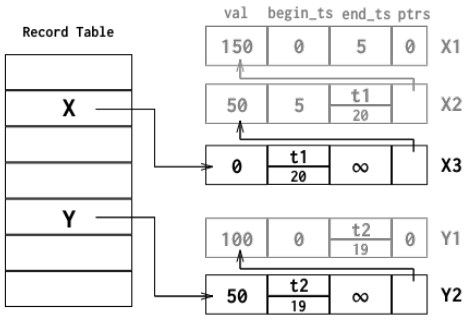


Figure 2: Example from Figure 1 shown in storage

tamp. This is the default case (and ultimately what we want when reading a version) where the version is considered "visible" and is read by the transaction. It is worth noting that when we initialize the database (as well as add any new version to a record), we set the end timestamp of the latest version of any record to ∞, which makes this default case fairly common during transaction execution, especially during low contention.

The second and third read cases require further checking than the default case. For example, if the version being considered by the transaction was inserted by another transaction (T2) that has not yet committed, the version is considered "invisible." In our implementation, this is noticed when the begin timestamp of a version points to a transaction instead of an actual timestamp number. In this case, the storage manager first checks if T2 is in its COMMITTED phase, and the version is read by the transaction if T1's begin timestamp is greater than T2's end timestamp. Otherwise, if the transaction is still ACTIVE, then the version is deemed invisible to T1 and the storage manager continues to search for an older version that is "visible" to T1.

The final possibility is if the version's end timestamp, instead of the begin timestamp, points to a running transaction (T2) and not a timestamp id. This happens when another transaction is currently writing to that version. In this case, we once again check that the transaction is not ACTIVE (or this version is "invisible"). Otherwise, the end timestamp is read from T2's end timestamp, and the storage manager checks that T1's begin timestamp is between that version's begin and end timestamps.

When initializing writes to the storage, the cases to check for are far simpler, and only attempt to ensure that there are no write-write conflicts. In the case the manager determines that a write-write conflict will occur, the transaction is aborted and restarted. Otherwise, the new version is pushed onto the top of the record's version list, and the transaction handles finishing its execution.

4.3. Transaction Execution

After the original reading from storage, a transaction executes by reading and writing the necessary values from its local copies. During transaction execution, we construct our constraint path which is a path down a decision tree where the decisions are the results of our constraint checks. This means that it is not required that all constraints pass, but that the path taken down the decision tree at validation time is the same as the one taken at execution time.

After execution, as mentioned above in 4.1, a transaction obtains an end timestamp and reads its constraint set from storage before entering the validation stage. During validation, a transaction traces down the path of the constraint decision tree once again. If at any point it diverges from the path taken during run time, then the transaction is signaled to abort. It is important to note that a transaction's *Read()* function executes slightly differently during validation. When not in validation, a transaction that reads a record that it itself wrote will read the latest version (i.e. the local copy of the transaction). During validation this feature is turned off since we want to compare a transaction's read records with those (possibly modified) records in the current database, not with those written by the transaction. In other words, at validation time a transactions *Read()* function reads from its constraint set (the results from re-reading the database after execution), not from the set read at the beginning of the transaction.

In our current implementation, we only support binary decision trees. To support multi-decision constraint paths is future work. In addition, the list of constraints is obviously application dependent, thus it is important that the DBA *a priori* have constraints to limit transactions. As in [6], it is possible to integrate those constraints at compile time by the compiler. This is not done in our current implementation, but the responsibility can be placed on the compiler to take a list of constraints and create instructions that will create logic such that a data structure can be built to check against during validation.

As described in 3.2, at run time we construct a constraint path down our decision tree. To do this, we keep a vector of the results from a series of *if/elseif* statements that contain our constraint checks. At validation, we attempt to get the same vector using the same series of *if/elseif* statements. We commit only if the vectors are the same, otherwise we fail to validate and proceed as described in 4.1.

5. Experimental Evaluation

Presented below are our experimental results after evaluating the constrained snapshot isolation level against a standard snapshot isolation level [9] and a stan-

dard serializable level [1].

5.1. Revised Smallbank Benchmark

In order to evaluate the performance of CSI, we constructed a revised version of the SmallBank benchmark proposed in [4]. It assumes that our database has two tables, a CHECKING and SAVINGS table. We implemented two types of transactions: *WriteCheck* (WC) and *WithdrawSavings* (WS). Both types of transactions contain a predefined constraint built into the transaction type (in a complete system this could be inserted at compile time by the compiler). For both types, the constraint is that the sum of CHECKING and SAVINGS for a given key must be greater than C, where C is a given bound. If the constraint holds, *WriteCheck* transactions withdraw the given amount from the CHECKING account. If the constraint fails, *WriteCheck* transactions deduct the withdrawal amount plus a penalty from the CHECKING account. *WithdrawSavings* transactions act in a similar fashion but for the SAVINGS table instead.

We are able to simulate write-skew with just these two transaction types by running them on a compact set of records. Under these workloads, we get write-skew when a WS and WC transaction concurrently edit the same customer's CHECKING and SAVINGS accounts. Similar to the example in section 3.1, if each transaction (one of type WS and the other WC) decrements the balance in both the CHECKING and SAVINGS tables and breaks the initial constraints, the result is a database state that is the result of a non-serializable set of transactions. In that case, our CSI implementation catches the write-skew and restarts whichever transaction finishes later.

5.2. Results

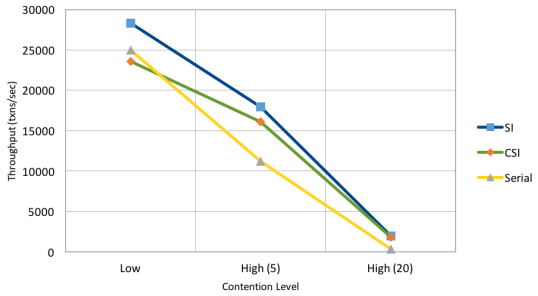
Experiments were performed on the Yale Zoo cluster (zoo.cs.yale.edu). In particular, experiments were run with care taken to monitor resource usage such that the machines were not already under heavy usage. In general, each Zoo machine has eight processors, four cores per processor and 32GB of memory.

Experiments were run by executing a load generator that created batches of transactions that accessed or updated randomly chosen keys in the database. We tested the performance of CSI, SI, and a serial MVCC with just *WriteCheck* (WC) transactions, just *WithdrawSavings* (WS) transactions, and a 50/50 mixture of *WriteCheck* and *WithdrawSavings*. Under each of these conditions we tested a "low contention" version with a database size of one million entries and a"high contention" version with a database size of one hundred entries. Each of the low and high contention load tests also tested the performance under transactions that access 5 records in the database and transactions that access 20 records in the database. Finally, to accentuate the difference in performance between the different test scenarios, we also tested the implementation under different transaction durations, simulated by letting the transaction spin for 0.0001, 0.001, and 0.01 seconds each. Each data point obtained below is an average of three separate runs.

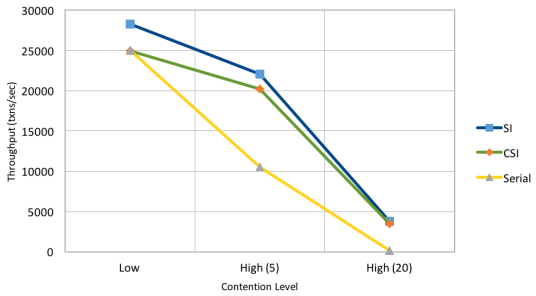
Figure 3.1 shows the results of running only WC or WS transactions. This implies that there is no possibility that write-skew will happen. This is used as a preliminary proof-of-concept to make sure that CSI outperforms a locking concurrency control system and that it approaches SI for high contention levels. An interesting note is that for very low contention (each transaction accesses only 5 records out of 1 million in the database) locking seems to do better than CSI. This is possible because either: 1) the overhead we use for validation is slightly larger than the overhead used for locking in low contention, or 2) the extra reading from storage for validation in CSI causes slight latencies that are not present in the serial version.

Figure 3.2 below shows a similar setup as Figure 3.1 but now we mix WC and WS transactions such that write-skew becomes possible. Every time there is a conflict we are guaranteed to have a possibility of write-skew in SI. This is the reason for CSI to perform slightly worse than SI. For those write-skewed transactions that break the given constraints, CSI would abort but SI would commit both transactions. We see that between Figure 3.1 and 3.2, the performance of SI and CSI actually improves for the higher contention levels. For SI, we see this performance increase because from Figure 3.1 to 3.2 we halve the number of transactions for the corresponding type used in 3.1. Specifically, if we used 100% of transactions to be WC in the no write-skew experiment, then 50% would be WC and 50% WS for the write-skew experiment. Because of this, we would expect the write-skew experiment to have fewer write-write conflicts (since only transactions of the same type can have write-write conflicts), thus resulting in higher throughput for higher contention levels for SI. For CSI, a similar situation occurs. A higher percentage of transactions make it to the validation phase (since fewer number of same-type transactions result in fewer write-write conflicts) and subsequently pass validation. Thus we see an increase in CSI write-skew performance from the no write-skew experiment.

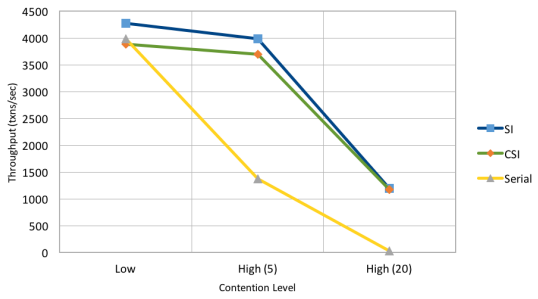
In evaluating longer transactions—i.e. transactions with logic that lasts 1ms and 10ms, as shown in Figure 3.3 and Figure 3.4 respectively—the performance gap between serial MVCC and both SI and CSI grows larger. This is partially the result of serial MVCC wasting transactional logic, even for transactions that may conflict and abort. Note that in serial MVCC, writes conflict with reads, which is not the case in either SI or CSI.



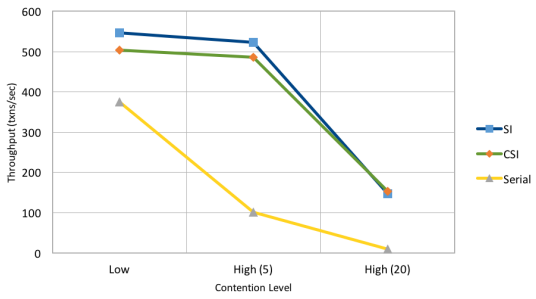
1. No Write-Skew



2. With Write-Skew



3. Long Transactions (1ms) with Write-Skew



4. Long Transactions (10ms) with Write-Skew

Figure 3: 3.1 (Top Left) shows throughput without write-skew on short transactions. 3.2 (Top Right) shows throughput with write-skew on short transactions. 3.3 (Bottom Left) shows throughput with write-skew on 0.001-second transactions. 3.4 (Bottom Right) shows throughput with write-skew on 0.01-second transactions.

Instead, these two mechanisms only abort in the face of write-write conflicts, and they do so before the execution of transaction logic (CSI might also abort during validation). Thus, they maintain relatively high throughput as compared to serial MVCC.

6. Related Work

Previous work on different isolation levels provides the main motivation for this work. Larson et. al [9] provides a framework for the implementation of SI and other weaker/stronger isolation levels. The serial abstraction that they use relies on keeping track of commit dependencies which should perform similar to our lock-based approach since dependencies among transactions cause locking in our version, and speculative blocking in their serial system.

Fekete et. al [7] is one of the first looks into the problems of SI. It explores the possible write-skew scenarios and poses the possible solution of serializing SI.

Cahill et. al [5] is inspired by SI and builds a serializable isolation level mechanism on top: SSI. It serves as the introduction of SSI which takes the snapshot isolation level and introduces anti-dependencies to check for possible cycles among transactions. It aborts if any anti-dependency is found. This is another way by which to solve the write-skew problem.

Cahill's thesis [4] provides the framework for the SmallBank benchmark that we implemented. In addition to fully explaining SSI, it shows under what situations (and types of transactions) write-skew happens and how SSI solves it.

7. Future Work

Future extensions of the work involve adding more test scenarios that better evaluate how CSI functions under different constraints. We intend to expand the types of transactions in our experiments by adding additional write-skew interactions to see how CSI fares in contrast with other concurrency control mechanisms under concurrent, varied write-skew transactions. This is different than our current evaluation method, which uses one pair of write-skew transaction types: *WriteCheck* and *WithdrawSavings*. In addition, our current implementation of CSI occurs entirely in main memory, inducing no disk access delays. Thus, it would be informative to evaluate CSI on a database implementation with data on disk to

see the effect of disk latency. Instead of forcing transactions to spin for a while to simulate transaction operations, it would be more realistic to have real transaction logic and operations in the test platform. An additional implementation of SSI would allow for direct comparisons between CSI and SSI.

Furthermore, we intend to improve our implementation of CSI in the following ways. First, we intend to remove the dependency on a single thread to schedule transactions to the worker thread. This causes an artificial bottleneck that affects all concurrency control mechanisms tested. The single transaction scheduler thread would be replaced entirely by parallel transaction threads that would execute pending transactions round robin style. To further increase the realism of the CSI system, we intend to incorporate garbage collection of old versions and a logging / recovery mechanism that allows for redundancy in case of failure by squirrels [10] or weasels [3].

8. Conclusion

Current multi-versioned database isolation levels provide either high performance on high contention mixed workloads or full serializability of transactions. In contrast, constrained snapshot isolation (CSI) guarantees stable constraint logic and provides high performance in high contention mixed workloads. Our design of CSI builds upon snapshot isolation by implementing a validation method, in which we reconstruct the transaction's control flow to ensure stable constraint logic.

The experiments of this new isolation level and concurrency control scheme show that CSI performs almost as well as snapshot isolation and significantly better than the serializable version, particularly in high contention with the presence of write-skew. Additionally, as the transactions increase in length, serializable isolation further decreases in performance, while CSI stays close to snapshot isolation. CSI seems to be both unique and practical, and this paper serves as a proof-of-concept for future work on this concurrency control mechanism.

References

[1] Abadi Daniel, Kun Ren, Faleiro Jose. Assignment 2. CPSC 438 Database Systems and Implementation. Spring 2016.

[2] Adya, Atul. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Diss. Massachusetts Institute of Technology. Cambridge, Massachusetts, 1999.

[3] Brumfiel, Geoff. "Weasel Apparently Shuts Down World's Most Powerful Particle Collider." *NPR*, 2016.

[4] Cahill, Michael James. "Serializable Isolation for Snapshot Databases." Thesis. The University of Sydney, 2009. Web.

[5] Cahill, Michael J., Uwe Rhm, and Alan D. Fekete. "Serializable Isolation for Snapshot Databases." *ACM Transactions on Database Systems (TODS)*, 34.4(20), 2009.

[6] Diaconu, Cristian, et al. "Hekaton: SQL server's memory-optimized OLTP engine." Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013.

[7] Fekete, Alan, et al. "Making snapshot isolation serializable." *ACM Transactions on Database Systems (TODS).* 30.2: 492-528, 2009.

[8] Hellerstein, Joseph M., Michael Stonebraker, and James Hamilton. *Architecture of a database system*. Now Publishers Inc, 2007.

[9] Larson, Per-Ake, et. al. "High-Performance Concurrency Control Mechanisms for Main-Memory Databases." *Proceedings of the VLDB Endowment.* 5 (4), 2011.

[10] McMillan, Robert. "Guns, squirrels, and steel: The many ways to kill a data center." *Wired*, 2012.