Transaction Repair for Multi-Version Concurrency Control

Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch

EPFL DATA Lab, Switzerland {firstname}.{lastname}@epfl.ch

ABSTRACT

The optimistic variants of Multi-Version Concurrency Control (MVCC) avoid blocking concurrent transactions at the cost of having a validation phase. Upon failure in the validation phase, the transaction is usually aborted and restarted from scratch. The "abort and restart" approach becomes a performance bottleneck for use cases with high contention objects or long running transactions. In addition, restarting from scratch creates a negative feedback loop in the system, because the system incurs additional overhead that may create even more conflicts.

In this paper, we propose a novel approach for conflict resolution in MVCC for in-memory databases. This low overhead approach summarizes the transaction programs in the form of a dependency graph. The dependency graph also contains the constructs used in the validation phase of the MVCC algorithm. Then, when encountering conflicts among transactions, our mechanism quickly detects the conflict locations in the program and partially re-executes the conflicting transactions. This approach maximizes the reuse of the computations done in the initial execution round, and increases the transaction processing throughput.

1. INTRODUCTION

Recent research proposes an optimistic MVCC algorithm as the best fit for concurrency control in in-memory databases [25]. This algorithm, like its predecessors [21, 15], aborts and restarts the conflicting transactions, which is simple but sub-optimal. Any conflict among transactions results in more work for the concurrency control algorithm and an increase in the execution latency of the transactions. Increased latency not only affects the throughput of individual transactions, but also increases the probability of having more concurrent transactions in the future. This might incur even more conflicts, forming a negative feedback loop.

The sub-optimality of the abort and restart approach becomes more significant when the number of conflicting transactions is high. The two factors that contribute the most to an increase in the number of conflicts are: (1) having high contention data objects that are read and updated by several concurrent transactions, and (2) having long running transactions, the lifespan of which intersects that of many other transactions.

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA

DOI: http://dx.doi.org/10.1145/3035918.3035919

In this paper, we introduce a novel multi-version timestamp ordering concurrency control algorithm, called Multi-Version Concurrency Control with Closures (MV3C). This algorithm resolves the conflicts among concurrent transactions by only partially aborting and restarting them. The main challenges for this type of conflict resolution technique are having: (1) a low overhead on the normal execution of transactions, as every transaction pays this cost irrespective of encountering a conflict, and (2) a fast mechanism to narrow down the conflicting portions of the transactions and fixing them, as a mechanism that is slower than the abort and restart approach defeats the purpose. The first challenge is dealt with by reducing additional bookkeeping by reusing the existing concurrency control machinery. To address the second challenge, MV3C uses lightweight dependency declaration constructs in the transaction programs that help in immediately identifying the dependencies among different operations. Using this dependency information, MV3C pinpoints blocks of the program affected by the conflicts and quickly re-executes only those blocks. The dependency information is added either manually by the user, or by employing static program analysis and restructuring.

The rationale for proposing MV3C is that a conflict happens only if a transaction reads some data objects from the database that become stale by the time it tries to commit. Here, the assumption is that all data modifications made by a transaction are invisible to other transactions during its execution. These modifications become visible only after the critical section during which the transaction commits. Consequently, just before committing a transaction, by checking whether its data lookup operations read the most recent (committed) versions of the data objects, serializability of the execution is ensured. In addition, by associating read operations with the blocks of code that depend on them, the portion of a transaction that should be re-executed in the case of a conflict is identified quickly, and gets re-executed. These blocks correspond to a specific class of sub-transactions in the nested transaction model. The boundaries of these blocks are specified by MV3C, which makes it possible to efficiently repair conflicting transactions. This is further discussed in section 7.3.

Our experimental results show that MV3C can achieve more than an order of magnitude in transaction processing throughput for high-contention scenarios compared to optimistic MVCC. Specifically, under high-contention, MV3C achieves around twice the throughput of optimistic MVCC for the well-known TATP[37] and TPC-C[31] benchmarks. In particular, this paper makes the following three contributions:

1. An efficient conflict resolution mechanism for multi-version databases, MV3C, which repairs conflicts instead of aborting transactions, with minimal execution overhead. The design of this mechanism is discussed in section 2.

2. A method to deal with write-write conflicts in MV3C. Unlike other optimistic MVCC algorithms, MV3C can avoid aborting the transaction prematurely when a write-write conflict is detected. This is discussed further in section 2.3.1.
3. A mechanism for fixing the result-set of failed queries for MV3C, which can optionally be enabled for each query in a transaction. This mechanism can boost the repair process for transactions that have long running queries. The details of this mechanism are described in section 4.2.

Motivating examples. The three main cases of transaction programs that benefit from MV3C are illustrated in Figure 1. Each case in this figure shows an instance of a transaction program starting with a *begin* command and finishing with a *commit* command. Each program consists of one or more blocks of code represented by a box. The dependencies among different blocks of code are represented by arrows. If block Y depends on block X, there is an arrow from X to Y. Moreover, it is assumed that each of these three instances failed during its validation phase, because of a conflict detected in its block A.

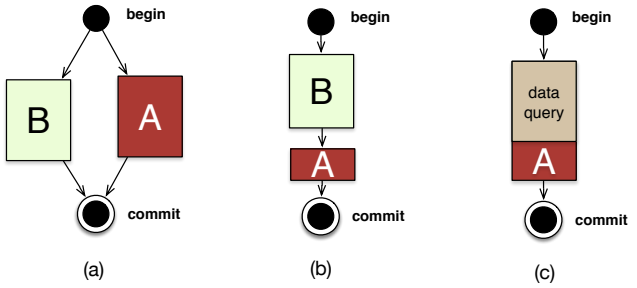


Figure 1: Cases where MV3C is more efficient in repairing the conflicting transactions compared to the "abort and restart" approach.

In the following, the three scenarios shown in Figure 1 are described using examples and the approach of MV3C for resolving the conflict is explained for each case.

First. In this case, the transaction has logically disjoint program paths and conflicts happen only in a few of them. Such paths are detected from the program structure and only they are re-executed. One example of this case is shown in Figure 1(a).

Example 1. Assume that in Figure 1(a), block A reads a row from table T_A and updates it, and block B reads a row from table T_B and updates it. Moreover, these updates only depend on the input parameters of the program. Then, if a concurrent transaction also updates the same row from table T_A and commits before the other transaction, the latter fails to commit. However, MV3C detects that only block A has a conflict and re-executes only this block, without re-executing block B. △

Second. In this case, conflicts happen after doing a substantial amount of work in the transaction. Here, it is not necessary to redo all the work. Instead, the data available before the conflict is reused in order to continue from the point of conflict. Figure 1(b) shows this case, where only the conflicting block A is re-executed. A concrete example of this case is the Banking example described in Example 2. In the following sections, this example is used in order to better describe MV3C.

Example 2. **Banking example:** The example consists of a simplified banking database with an Account table. This table stores the balance of the customers identified by an ID. There are two types of transaction programs that run on this database. The first program, named SumAll, is read-only. A SumAll transaction sums up the balances in all the existing accounts. The second program is named TransferMoney and it is written in a PL/SQL-like language as shown in Figure 2. A TransferMoney transaction transfers a specific amount of money from one account to the other, given the availability of sufficient funds. The money transfer also consists of a fee that is deducted from the sender account and is added to the central fee account identified by FEE_ACC_ID.

Now, assume that two TransferMoney transactions, using different input parameters, run concurrently. Then, the first one succeeds, and the other one fails due to line 17 in Figure 2. MV3C detects that only the operation that reads the current value of the fee account impacts the correctness of line 17. Thus, only that line gets re-executed, this time, with the new value of the fee account. △

Third. In this case, conflicts occur in the beginning of the transaction. Then, the data returned by SELECT queries to the database is reused after accommodating the changes introduced by the conflicting transaction(s). Thus, the re-evaluation of queries from scratch is avoided. As illustrated in Figure 1(c), even though the transaction program consists of a single block of code, the initial part of the block responsible for querying the data from the database is re-executed more efficiently under MV3C.

Example 2 (continued). In the Banking example, assume that there is another transaction program, named Bonus. This transaction program increments the balance of the accounts with a minimum balance of 500 CHF by 1 CHF. As the *balance* column is not indexed, a Bonus transaction has to scan the whole Account table. Meanwhile, a concurrent TransferMoney transaction commits, increasing the balance of an account above 500 CHF. Then, the Bonus transaction fails validation, as it did not consider the new record in the Account table. However, as MV3C knows that the conflict happened only because of that record, it fixes the result-set of the query by including the additional record. This completely avoids another round of full scan over the Account table. △

2. MV3C DESIGN

The main idea behind MV3C is that by exposing the program dependencies to the concurrency control algorithm, conflicts among concurrent transactions can be resolved efficiently. This information is already provided to the transaction processing system via user defined transaction programs [9, 30, 40]. However, an abstract view of the program is needed to exploit this dependency information. In this abstract view, correctness checks are associated with blocks of code. Then, each executed instance of the transaction program, referred to as a *transaction*, uses this information in order to recover efficiently from a failure due to a conflict.

For this purpose, the possible failure points are identified and the transaction program is partitioned into smaller blocks such that each failure is contained within a single block. These program blocks are independent, and the failure of some blocks do not affect the other blocks in any way. The failure possibility stems from having a predicate in that block of the program, which might not pass the validation phase. After a failed validation, a new timestamp is assigned to the transaction, but only those blocks that have an invalid predicate inside them are rolled back and re-executed. The validation semantics guarantees that the other predicates would return the same values as the initial execution, and therefore, re-executing them is unnecessary.

In the rest of this section, we describe the design of MV3C. MV3C builds its efficient conflict resolution mechanism on top of the algorithm proposed in [25]. Throughout the rest of this paper, this algorithm is referred to as OMVCC, where O stands for *Optimistic*. A brief overview of OMVCC is provided before going into

```
/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
 START;
 SELECT bal INTO :fm_bal FROM Account WHERE id=:fm_acc;

 IF(amount < 100) fee = 1.0;
 ELSE fee = amount * 0.01;

 IF(fm_bal > amount+fee) {
  SELECT bal INTO :to_bal FROM Account WHERE id=:to_acc;

  fm_bal_final = fm_bal - (amount + fee);
  to_bal_final = to_bal + amount;

  UPDATE Account SET bal=:fm_bal_final WHERE id=:fm_acc;
  UPDATE Account SET bal=:to_bal_final WHERE id=:to_acc;
  UPDATE Account SET bal=bal+:fee WHERE id=:FEE_ACC_ID;
  COMMIT;
 } ELSE ROLLBACK;
}
```

Figure 2: TransferMoney transaction program from the Banking example (Example 2) in a PL/SQL-like language.

the details of MV3C, as the latter borrows some features from the former.

2.1 OMVCC overview

OMVCC gathers predicates for all the read operations of a transaction. A predicate in OMVCC can be thought of as a logical condition created using the attributes of a relation, encapsulating a data selection criterion. For example, the WHERE clause in a SELECT statement on a single table is a predicate for that table. The candidate predicates in our example program are highlighted in Figure 2. Moreover, OMVCC makes a reasonable assumption that every transaction writes into only a limited number of data objects. Therefore, it is feasible to keep track of the write-set of a transaction (i.e., the *undo buffer*) during its execution.

As OMVCC is an optimistic algorithm, it requires a validation phase before a successful commit. A variant of *precision locking* [16] is used for validating transactions. This approach requires that the result-sets of all the read operations are still valid at commit attempt time, as if the operations were done at that time. This creates an illusion that the whole transaction executed at commit time.

In order to achieve this, all committed transactions are stored in a list called *recently committed* transactions. During the validation phase of a transaction T, all of its predicates are checked against all committed versions of the transactions in the recently committed list. From this list, only those transactions that committed during the lifetime of T are considered. The undo buffers of these (concurrently executed) transactions contain the changes of this time period. If any committed version satisfies one of T's predicates, then the data read by the transaction is obsolete. The newer version should have been read instead, if the read operation used the commit timestamp as its reference point. In this case, T fails validation, rolls back and restarts.

2.2 MV3C machinery

MV3C gathers the predicates used during the execution of transaction, similar to OMVCC. Moreover, MV3C creates a new version for each modification of a data object. Each data object keeps a list of versions belonging to it, called its *version chain*. When a version is created for a data object, it is added to the head of this chain. The notion of *version* is defined below.

Definition 2.1. *A **version** is a 4-tuple (T, O, A, N), where T is either the commit timestamp of the transaction that created the version, or the transaction ID if the transaction is not committed, O is identifier of the associated data object, A is the value of O maintained in this version, and N is a version identifier if more than one version is written by T for O.*

The value written in a version is immutable. It cannot be modified, even by its owner transaction. Given a version V, the version identifier N in V is used to differentiate distinct versions written by the same transaction for the same data object. However, after a transaction is committed, only the newest version written by the transaction becomes visible to the other transactions. In practice, N can be a pair of pointers, pointing to the older and newer versions in the internal chain of versions written by a single transaction. Then, the committed version is the one without a newer version in N. The notion of a committed version is defined below.

Definition 2.2. *A **committed version** is a triple (T, O, A), where (T, O, A, N) is a version such that N is the identifier of the latest version of O written by the transaction with commit timestamp T.*

The read operations in MV3C traverse the version chain until a visible version is reached. A visible version either is owned by the transaction itself or is the latest committed version before the transaction started.

Definition 2.3. ***Visible version:** a version (T_1, O, A, N_1) is visible to a transaction with start timestamp T_2 if either:*

- *T_1 is committed, $T_1 < T_2$ and there is no other version $(T_3, O, \_, \_)$ where $T_1 < T_3 < T_2$, or*
- *$T_1 = T_2$ and there is no other version $(T_1, O, \_, N_2)$ where N_2 is newer than N_1.*

MV3C transactions, like those in OMVCC, have an undo buffer that maintains the list of versions created by the transaction. After a transaction commits, its undo buffer contains only the committed versions. The undo buffer is a representative of the effects of a committed transaction, as it contains all the modifications done by the transaction to the multi-version database. In addition, each predicate has a closure bound to it. The term *closure* is used in the sense of the programming languages literature [20]. In the rest of this paper, the term *predicate* refers to an *MV3C predicate*, unless otherwise stated.

Definition 2.4. *An **MV3C predicate** X consists of (1) a data selection criterion, (2) a closure $C(X)$ bound to it, (3) a list of versions $V(X)$ registered to it, and (4) a list of child predicates $D(X)$. $V(X)$ and $D(X)$ are populated after $C(X)$ is executed.*

Definition 2.5. **C(X)** *(**the closure bound to predicate X**) is a deterministic function that encloses all operations in a transaction program that depend on the result of X. $C(X)$ receives the result of evaluating X along with a set of context variables as its parameters.*

Example 2 (continued). The equivalent program of Figure 2 translated into MV3C DSL is illustrated in Figure 3. In Figure 3, the shaded part of P_1 is an MV3C predicate whose closure is the gray box underneath it. △

A closure can contain data selection or data manipulation operations apart from computations. Each data selection operation creates a new predicate with its own closure, which has access to its parent predicates and their result-sets. Moreover, each data manipulation operation (i.e., insert, delete and update statements) creates a new version of the data object.

MV3C requires specifying the relationships of the predicates to the versions and to the other predicates. A new Domain Specific Language (DSL), called MV3C DSL, is used in MV3C for writing transaction programs. MV3C DSL is meant to capture dependencies among operations inside a transaction program. Basically, it

is a simple library for encoding the relationships among the predicates, and binding closures to them. It also defines the granularity in which the algorithm operates, which is at the statement level by default. A naïve pessimistic translation of an existing program written in a PL/SQL-like language to the MV3C DSL requires a simple dependency analysis. One such translation can be derived by assuming that each operation in the transaction program depends on all of its previous operations. The same naïve approach can be used to execute ad-hoc transactions, where the whole transaction is not given to the system in advance, i.e., the ad-hoc commands of transaction are issued from an application program. However, program restructuring can be used along with detailed program analysis to generate more accurate dependency constructs. Further details about translating transaction programs written in a PL/SQL-like language to the MV3C DSL are described in Appendix B.

```
/* fm = from, acc = account and bal = balance */
TransferMoney(fm_acc, to_acc, amount) {
 START;

  IF(amount < 100) fee = 1.0;
  ELSE fee = amount * 0.01;

 P1 Account WHERE id=:fm_acc => fm_acc_entry
 IF(fm_acc_entry.bal > amount+fee) {
  fm_acc_entry.bal -= (amount + fee);
  fm_acc_entry.persist();

  P2 Account WHERE id=:to_acc => to_acc_entry
  to_acc_entry.bal += amount;
  to_acc_entry.persist();

  P3 Account WHERE id=:FEE_ACC_ID => fee_acc_entry
  fee_acc_entry.bal += fee;
  fee_acc_entry.persist();
  COMMIT;
 } ELSE ROLLBACK;
}
```

Figure 3: An example transaction program in the MV3C DSL.

To specify the relationships among predicates, when a predicate is created in the closure $C(X)$ of some predicate X, it is added to $D(X)$, the list of child predicates of X. For example, in Figure 3, P_2 and P_3 are added as children of P_1. A graph is constructed internally during the execution, using this parent-child relationship information. By definition, the predicates form a Directed Acyclic Graph (DAG), because there cannot exist any edge from a new predicate to an older predicate and hence a cycle cannot be formed. More intuitively, a SELECT statement cannot be executed if it requires a parameter that is available only after executing another SELECT statement in the future. Consequently, the result of executing a transaction program in MV3C is a DAG of predicates with a closure assigned to each predicate. The DAG for a transaction T is called the *predicate graph of* T. In addition, to specify the relationships between predicates and versions, references to the newly created versions are stored into $V(X)$, the list of versions registered to X. Then, using these references, X keeps track of the created versions that directly depend on its result-set.

Apart from the data objects that are altered in the database, there can also be mutable variables in a transaction program. Each mutable variable is local to a closure, as sharing it among more than one closure makes it impossible to reason about the state of the variable if any of these closures is re-executed. However, an immutable copy of the variables defined in a closure is shared with the closures of its child predicates. If a mutable variable is shared and modified among different closures, it would be treated as a database object and multi-versioning will be applied to it.

For $C(X)$, the context variables are those variables that are defined outside the closure and are accessible to it. The context variables are immutable and are categorized into: (1) the input parameters of the transaction, (2) the result-sets of the ancestor predicates of X, and (3) the variables defined in the closures bound to the ancestor predicates of X. For (2) and (3), MV3C guarantees that if either the result-sets of the ancestor predicates or the variables defined in their closures change, operations of $C(X)$ are undone.

Example 2 (continued). In Figure 3, there are three predicates, P_1, P_2 and P_3, where P_2 and P_3 depend on P_1, as they are defined inside the closure of predicate P_1. Each predicate in this example has a closure that is represented as a gray box underneath it. The orange shade is the predicate itself, and the arrow symbol (=>) in front of each predicate represents the execution of the predicate, which returns the result of the evaluation, i.e., a copy of the latest visible version of the requested data object. Then, the returned version is stored in a variable, which is used inside the closure that is bound to the predicate. Inside the closure, the fields of the returned version are accessed and modified. In the case of a modification, a call to the *persist* function is necessary in order to store the changes into a new version inside the database, and adding it to the undo buffer of the transaction, as well as the list of versions of the predicate. △

Predicates are first class citizens in MV3C. Various types of predicates can be defined, all of which implement the predicate interface comprising an *execute* function and a *match* operation. The execute function accepts a closure as its parameter and binds it to the predicate. This function evaluates the predicate by collecting the visible versions that satisfy the data selection criterion into the result-set of the predicate. Then, the closure is executed with the result-set and the set of immutable context variables as its parameters. The match operation of a predicate checks whether a given version satisfies the selection criterion. It is used during the validation phase of a transaction. The lifecycle of an MV3C transaction is shown in Figure 4. The details of the execution, validation and repair phases of a failed transaction are described in sections 2.3-2.5.

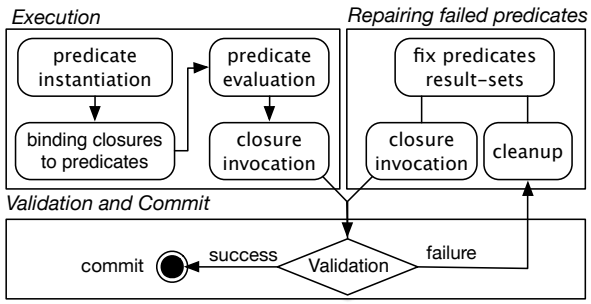


Figure 4: The life cycle of a transaction running under MV3C.

2.3 MV3C execution

An MV3C program starts by instantiating the root predicates of the predicate graph and calling their execute functions. Each root predicate is evaluated and the closure bound to it is executed. This instantiates the predicates defined in the closure, which are evaluated, and in turn, their closures are executed.

Example 2 (continued). Figure 5 illustrates a snapshot of the Banking example run under MV3C. P_1'', P_2'' and P_3'' are the predicates used in T_z. They are the runtime instances of predicates P_1, P_2

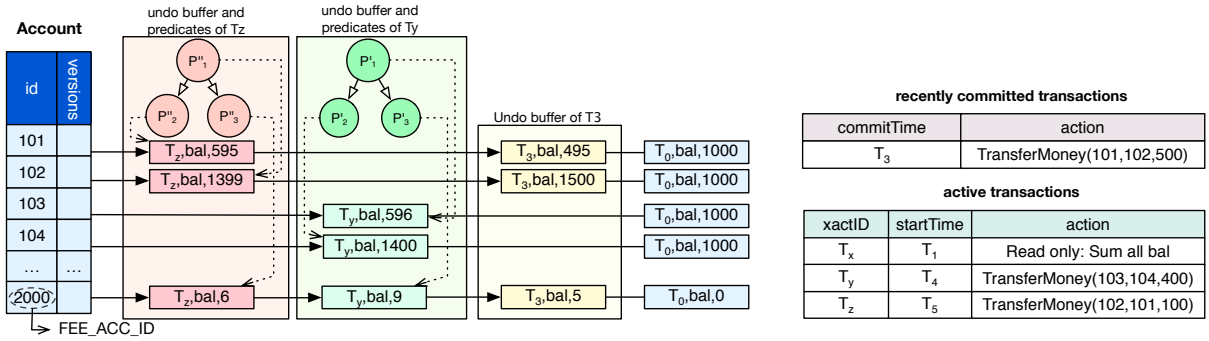


Figure 5: A snapshot of the MV3C database for the Banking example.

and P_3 shown in Figure 3. The dotted arrows represent references to their corresponding created versions. △

A transaction finishes its execution when the calls to the execute functions of all the root predicates return successfully. However, some transactions may abort prematurely. One reason is a transaction issuing an abort command after detecting an unwanted state. In this case, repairing or restarting the transaction is irrelevant. The rollback is performed and all the versions created until that point are discarded. Another reason for a premature abort is having a write-write conflict, described next.

2.3.1 *Handling multiple uncommitted versions*

A transaction can be involved in one or more read-write, write-read or write-write conflicts. Among these conflict types, optimistic MVCC algorithms deal with read-write and write-read conflicts during the validation phase. However, handling a write-write conflict is different. A write-write conflict happens when a transaction T tries to write into a data object that has another uncommitted version or a committed version that is newer than T's start timestamp. This type of conflict results in a guaranteed failure during the validation phase of MVCC algorithms such as OMVCC. As these algorithms do not have a method to recover from such conflicts, their best choice is to detect the write-write conflicts during the execution, abort the conflicting transaction prematurely and restart it.

Example 3. Consider a long running transaction which runs successfully until the last statement. The last statement writes the results into a highly contended data object. Rolling back the transaction just because of a conflict in this last statement might not be the best decision. On the other hand, if a write-write conflict is encountered in the first few statements of the program, and the rest of the program depends on the written data, then continuing the execution from that point would be a complete waste of resources. Here, the repair action to recover from this conflict is similar to restarting it from scratch. A conflict that happens in predicate P_3 shown in Figure 3 is of the former type, while one in predicate P_1 is of the latter type. △

For this reason, it is desirable to provide a configuration option for indicating how to handle write-write conflicts. The first option is to deal with a write-write conflict upon its detection by aborting and restarting the transaction early, as in [25]. If the programmer chooses this option, then the transaction involved in a write-write conflict is rolled back and restarted from scratch. Even though MV3C might still be able to save the work for the closures that are completely done before reaching the conflict, some additional book-keeping is required to make it feasible. For the purpose of keeping a low-overhead for MV3C, we deliberately abort and restart the transaction in this case as programmer indicates. Operations that modify the key or indexed fields, e.g., insert or delete are always treated this way and fail-fast if another uncommitted version exists in the version chain.

The second option to deal with write-write conflicts is to ignore the detection of the write-write conflict and proceed forward with creating a new version of the data object. Under MV3C, if multiple transactions are allowed to write into a single data object, each creates its own version and adds it to the version chain. Before committing any transaction that is involved in a write-write conflict, these additional versions are not important, as they are visible only to their own transactions. The approach taken in the validation and commit phase to handle this case without affecting serializability is described in section 2.4.1.

Example 2 (continued). In the snapshot shown in Figure 5, both T_z and T_y have concurrently written new versions for FEE_ACC_ID, because write-write conflicts were allowed in TransferMoney. Otherwise, if T_z executes the closure bound to P_3'' after T_y finishes all of its operations, T_z prematurely aborts and restarts, because of the uncommitted version written by T_y. △

We strongly believe that if there exists a method for conflict resolution of transactions (such as MV3C), premature abort and restart is not the best available option. Accordingly, the decision regarding this case should be made either by the transaction programmer or an automated program analyzer. In MV3C, the configuration of the write-write conflict behavior can be done as a system-wide or table-wide setting, which can be overridden for each individual update operation.

2.4 MV3C validation and commit

A transaction T that is not aborted prematurely has to be validated before it can be committed. The validity of T is dependent on the validity of its predicates. Other transactions that committed during the execution of T are potentially conflicting with it. Hence, it is necessary to check if any of the potentially conflicting transactions write into a data object that T read.

Definition 2.6. ***Valid predicate:*** *A predicate P belonging to transaction T with start timestamp S is defined to be valid at timestamp S' if and only if the result-set of P is the same when S' is used as the start timestamp of T instead.*

Proposition 2.1. *A predicate P belonging to transaction T is valid at timestamp C if there is no new version committed between S and C that satisfies P, where S is the start timestamp of T.*

Proof Sketch. The predicate P is evaluated on the snapshot of the database at S. The newly committed versions between S and

C exist in the undo buffer of the transactions committed in the same time period. These are the only changes that happened to the database in the same period. If P does not match any newly committed version, then there cannot be a change in its result-set. Thus, based on Definition 2.6, P is still valid at C. □

Accordingly, the validation phase consists of matching the predicates of the current transaction against the committed versions of potentially conflicting transactions. If none of the predicates match against any version, then the validation is successful, and the transaction commits. Otherwise, the matched predicates, along with all their descendant predicates, are marked as invalid. For this reason, the predicate graph is traversed in topological sort order, first traversing the higher-level predicates. The Validation algorithm (Algorithm 1) is used by MV3C to validate transactions.

Algorithm 1 Validation

```
1: Input: G(S) ← predicate graph resulting from executing a
                 transaction program T using start timestamp S
          S' ← validation timestamp, where S < S'
2: Output: L1 ← list of valid nodes in G(S) using S'
           L2 ← list of invalid nodes in G(S) using S'
                 along with their descendants
3: Q ← the result of applying topological sort on G(S)
4: while Q is non-empty do
5:     n ←remove the node from the head of Q
6:     if ((a parent predicate of n exists in L2) or
           (n is an invalid predicate at S')) then add n to tail of L2
7:     else add n to tail of L1
8:     end if
9: end while
10: return (L1, L2)
```

The Validation algorithm traverses the predicate graph in order to identify all invalid predicates and their descendants. This is different from OMVCC where the validation process stops after finding the first invalid predicate, as it cannot succeed anymore. By contrast, in MV3C the validation phase is responsible for identifying all invalid predicates. For this reason, the validation process resumes at the next predicate even though it is known that the transaction cannot succeed validation. This algorithm not only identifies the invalid predicates, but also topologically sorts and splits the predicate graph into two parts, L_1 and L_2. All valid predicates are in L_1, whereas the invalid predicates and their descendants are in L_2. The validation of a transaction, starting at timestamp S and resulting in a predicate graph $G(S)$, is successful at timestamp S', if L_2 in the result of Validation ($G(S)$,S') is empty, in which case the transaction commits with commit timestamp S'.

Example 2 (continued). In the snapshot shown in Figure 5, assume that T_y finishes execution first. Then, as there is no concurrent transaction that committed before T_y, it passes validation, its commit timestamp is assigned, and it is added to the recently committed list as T_6. Thereafter, T_z enters the validation phase and checks the recently committed list and finds T_6 as a concurrent transaction that committed after it had started its execution. Using the Validation algorithm, T_z traverses the predicate graph in topological sort order and matches each predicate against the committed versions of T_6. The Validation algorithm finds a match for P_3'' against $(T_6, bal, 9)$, making the predicate invalid. P_3'' is the only predicate in this example that gets added to L_2, the list of invalid predicates. △

Lemma 2.1. *L_1 in the result of the algorithm Validation ($G(S)$,S') (Algorithm 1) does not contain any invalid predicate nodes or the descendants of an invalid predicate node, for any given timestamps S and S', and predicate graph $G(S)$, where $S < S'$.*

Proof Sketch. This follows immediately from lines 6 and 7 of the Validation algorithm. □

Lemma 2.2. *$L_1 + L_2$, the concatenation of L_1 followed by L_2, is a topological sort of $G(S)$, where L_1 and L_2 are the results of the algorithm Validation ($G(S)$,S') (Algorithm 1).*

Proof Sketch. Algorithm 1 starts by topologically sorting the predicate graph. It then devides the result of the topological sort into two lists, L_1 and L_2. Since any subset of a topological sort is also topologically sorted, L_1 and L_2 individually follow the topological sort. To prove that $L_1 + L_2$ is topologically sorted, it is sufficient to show that no node in L_2 has a descendant node in L_1. In Algorithm 1, the only place where nodes are added to L_2 is line 6. Based on the condition in line 6 and the fact that nodes are traversed in topological sort order, if a node is in L_2, all of its descendant nodes are also in L_2. □

2.4.1 Impact of multiple uncommitted versions on validation

As explained in section 2.3.1, if write-write conflicts are allowed, multiple uncommitted versions can coexist under MV3C. Each version involved in a write-write conflict is the output of two possible scenarios. First, there is a predicate P in transaction T that reads the current value of the data object D before updating it. In this case, if another transaction T' with a version for D commits first, T fails validation while matching P against the committed versions of T'. Then, this conflict gets resolved in the Repair phase.

Second, the write is blind and the transaction updates the data object without reading its existing value for the updated fields. As a matter of fact, all writes done by a transaction, including the blind ones, are visible to the other transactions only after its commit. Note that the fields of the data object that are used for finding it in the table (i.e., key or indexed fields) do not affect the blind write property, as they remain intact. As it was mentioned in section 2.3.1, this assumption holds because the operations that modify the key or indexed fields are not treated as blind write operations and cannot interleave with other blind write operations. Therefore, accepting blind writes does not affect serializability, as the read-set of the transaction remains unmodified at commit time, and the illusion of running the entire transaction at commit time is maintained.

Example 2 (continued). Assume in the TransferMoney transaction, there is an extra input parameter named *date* that specifies the date and time of the transaction. Moreover, instead of giving a fee to the central account identified by FEE_ACC_ID, the date and time of the last transaction is updated using the *date* input parameter. That is, substituting the command on Line 17 of Figure 2 with the following SQL command:

```
UPDATE Account SET t_date=:date WHERE id=:FEE_ACC_ID;
```

This SQL statement translates into a blind write, as the data object is looked up using its primary key *id* and its value field *t_date* is updated without reading its current value. If the central fee account exists and execution is successful, this operation cannot fail in validation unless its predicate P_1 fails. △

Moreover, to keep the procedure for finding the visible value simple, MV3C moves a committed version next to the other committed versions in the version chain. This technique preserves the semantics that uncommitted versions are in the beginning of the

version chain, and committed ones are ordered by their commit timestamps at the end of the version chain. For example, in Figure 5, if T_z is committed before T_y, its versions are moved next to the versions created for T_3.

2.5 MV3C repair

If a transaction fails validation, it enters the repair phase. All the read operations in a transaction that run under a timestamp ordering algorithm (such as MV3C or OMVCC) return the same result-sets when re-executed, if the start timestamp does not change. Thus, a transaction that reads obsolete data, if re-executed, would read the same data and fail validation again. Therefore, the first step for repairing the transaction is picking a new start timestamp S' for it. The Repair algorithm, shown in Algorithm 2, is applied with the parameters ($G(S)$, S'), where $G(S)$ is the predicate graph resulting from the initial round of execution using start timestamp S. It uses the results of the Validation algorithm to prune the invalid predicates in the predicate graph.

Algorithm 2 Repair

```
1:  Input: G(S) ← predicate graph resulting from executing a
                  transaction program T using start timestamp S
           S' ← validation timestamp, where S < S'
2:  Output: G'(S') ← repaired predicate graph
3:  (L1, L2) ← Validation (G(S),S')
4:  F ← Set of all nodes in L2 with no incoming edges from L2
5:  for all predicate node f in F do
6:      for all predicate node h in descendant nodes of f do
7:          clear the list of versions in h and remove them from the
              undo buffer of the transaction
8:          remove h from G(S)
9:      end for
10:     clear and remove list of versions in f and remove them
          from the undo buffer of the transaction
11: end for
12: for all predicate node f in F do
13:     call f.execute(C(f), S'), where C(f) is the closure bound to f
14: end for
15: return G(S) as G'(S')
```

In line 4 of the Repair algorithm, the set of nodes in L_2 with no incoming edges from L_2 are selected into F, where L_2 is the second part of the output of the Validation algorithm. Based on the selection criterion, it is guaranteed that the nodes in F are not descendants of each other. It can be observed in lines 6 to 8 that a given invalid predicate f in F is *pruned*, which is the process of removing all the versions created by f and its descendants, and then removing the descendants from the predicate graph. After pruning the predicate node f, it is sufficient to re-execute $C(f)$, which is done in line 13. Executing the closure re-instantiates all pruned descendant predicates. The order of re-executing the invalid predicates does not matter, as they are independent. If one is a descendant of the other, the former would have been removed from the graph during the pruning process. In the Repair algorithm, the closures of the valid predicates are not re-executed, as there are no changes in their result-sets.

Example 2 (continued). After T_z fails validation, a new start timestamp T_7 is assigned to it. Based on the validation results, P_3'' is the only invalid predicate in T_z. P_3'' reads the version written by T_6 with $bal = 9$ and creates $(T_z, bal, 10)$ as its version. △

After applying the Repair algorithm, the transaction enters the validation phase again. The validation phase is the same as the one for the initial execution of the transaction and has a possibility of success or failure.

Example 2 (continued). In the above example, T_z succeeds in the validation phase this time, as there is no concurrent transaction committed during its new lifetime (after T_7). △

In the case of a validation failure in this stage, another round of repair is initiated and this cycle continues until validation is successful. It is important that the whole process of validating a transaction, and drawing a commit timestamp or a new start timestamp depending on the result of the validation is done in a short critical section. However, the repair phase is done completely outside the critical section, as if the transaction is running concurrently with other transactions.

The main purpose of Repair algorithm is rebuilding the same predicate graph as if it is aborted and restarted from scratch. Two predicate graphs are equivalent if there is a graph isomorphism between them and corresponding predicate nodes have the same list of versions attached to them. The following lemmas prove the correctness in behavior of the Repair algorithm.

Lemma 2.3. *Given a predicate graph $G(S)$ and an arbitrary node X in it, pruning X and re-executing $C(X)$, the closure bound to X, under the same start timestamp S, rebuilds $G(S)$.*

Proof. Observe that both execution and re-execution of $C(X)$ view the same snapshot of the database, as the start timestamp of the transaction is not changed. Since all the versions created by X and its descendants are pruned, any changes created by the closures bound to X and its descendants are removed. In addition, C(X), based on Definition 2.5, is a deterministic function, which guarantees that given the same input, it generates the same output. Therefore, re-executing $C(X)$ re-creates the same predicate graph. □

Lemma 2.4. *Let $G(S)$ be the predicate graph resulting from the execution of a transaction T with start timestamp S that failed validation at timestamp S'. Assume that $G'(S')$ is the predicate graph resulting from applying the Repair algorithm on ($G(S)$, S'). Instead of applying the Repair algorithm, if T was aborted and restarted with start timestamp S', resulting in a predicate graph $G''(S')$, then $G''(S')$ is equivalent to $G'(S')$.*

Lemma 2.4 is proven in Appendix A.

2.6 Serializability

We claim that the schedules created using MV3C are commit-order serializable. Consequently, it is guaranteed not only that the schedules created by MV3C are serializable, but also that one equivalent serial schedule can be proposed by creating a serial sequence of transactions using their commit order under MV3C. The serializability of MV3C is stated in Theorem 2.1 and it is proved in Appendix A.

Theorem 2.1. *The committed projection of any multi-version schedule H produced under MV3C is conflict equivalent to a serial single-version schedule H' where the order of transactions in H' is the same as the order of successful commit operations in H and uncommitted transactions are ignored.*

3. INTEROPERABILITY WITH MVCC

Interoperability is an important aspect of any emerging concurrency control technique. There are many database and transaction

processing systems that are already operating using an existing concurrency control algorithm. Therefore, it is highly desirable that a new concurrency control technique is interoperable with its predecessors, as it facilitates the incremental acquisition of the new algorithm into the system by different transactions. This decreases the cost and risk of employing the new algorithm.

The only interaction of an MV3C transaction with other transactions is during the validation phase. The algorithm needs to know about the versions committed during the lifetime of the current transaction, so as to match the predicates of the current transaction against those versions. This is the only piece of information that is required from a previously executed transaction. Consequently, any MVCC algorithm, such as OMVCC, that provides information about the committed versions can seamlessly inter-operate with MV3C. This interoperability provides backward compatibility for free, and makes it possible for the transaction developers to program their new transactions in MV3C, and gradually convert the old transactions in the system into MV3C.

4. OPTIMIZATIONS

Like MVCC, there can be different flavors of MV3C, each with its own optimizations. This section is dedicated to optimizations that can be employed by MV3C.

4.1 Attribute-Level Predicate Validation

To validate an MV3C transaction, the committed versions of all concurrent transactions are considered. These versions are matched (as the whole record) against each predicate in the predicate graph. If a version satisfies a predicate, a conflict is declared. However, it is a pessimistic approach, the modified columns in the concurrently committed versions might not be used in the current transaction.

As an optimization, the validation can be done at the attribute-level instead, as in previous works [25]. To enable the attribute-level validation, the columns that are used from the result-sets of the predicates are marked for runtime monitoring. In addition, all columns in the data selection criterion of the predicate are monitored. At runtime, each created version stores a list of columns modified in it. Then, in the validation phase, while checking a predicate against a version, the intersection between the monitored columns in the predicate and the modified columns in the version is first computed. If the intersection is empty, there cannot be a match. Otherwise, the predicate-specific match operation is performed.

4.2 Reusing Previously Read Versions

The predicates that failed validation are the starting points of the repair phase. After acquiring a new start timestamp in this phase, the failed predicates are re-evaluated, and their result-sets are fed into their assigned closures. Evaluating these predicates from scratch is one approach and it is taken by default in MV3C. However, it is also possible to re-use some of the computation done in the initial execution round that failed validation.

The realization of this optimization depends on the type of predicates and the way they access their target data. However, there are some similarities among all of them. Each predicate that wants to re-use the computation keeps a reference to its result-set. The result-set has to be computed nonetheless, as it needs to be fed into the closure of the predicate. This additional reference is kept only until a successful commit, which is also the end of the lifetime of the predicate. Then, in case of a failed validation, the result-sets have to be fixed for the failed predicates, a procedure that is predicate-specific. This procedure is merged with the validation phase, as both require matching predicates against versions.

Example 4. Consider a predicate that selects data based on a condition over non-indexed columns of a large table. The initial execution is costly, as a full scan over the table is required. However, if this predicate fails, it must be due to a concurrent transaction committing a version that should be a part of the result-set of the predicate. Therefore, the result-set can be fixed by accommodating the concurrently committed versions into the result-set. △

This optimization comes with the cost of keeping a reference to the result-set of each predicate, as well as the procedure for fixing them. Therefore, it is not used by default, and can be enabled per predicate instance. The decision to use this optimization is taken either by the transaction programmer, or an automated analyzer that monitors the failure rate of each predicate. One heuristic is to activate this optimization only for predicates that have a higher failure probability, where fixing the result-set is cheaper than re-evaluation.

4.3 Exclusive Repair

In principle, the repair phase of MV3C (cf. section 2.5) happens concurrently with other transactions. This design decision can increase the parallelism degree of MV3C, as there is no guarantee about the time required for the repair phase. However, this strategy requires another round of validation after the repair phase is done. As an optimization, it is possible to apply the repair phase in a critical section that prevents other transactions from committing, even though it does not prevent them from running or validating. Thus, an extra validation round can be avoided and the transaction is guaranteed to commit after the repair phase. This optimization should be applied with the outmost care, as it can become a bottleneck in the system. A heuristic is to apply this optimization after N rounds of validation failures after applying the repair phase if the previous repair phases were short.

5. IMPLEMENTATION

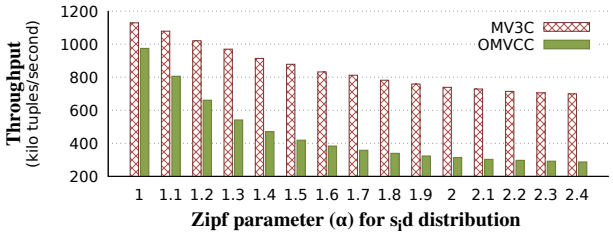
This section describes the details of our implementation.

Transaction Management. MV3C has a *transaction manager* that is responsible for starting and committing the transactions, similar to the one in [25]. It primarily stores four data items which are shared among all transactions, namely *recently committed* transactions, *active* transactions, *start-and-commit timestamp sequence generator*, and *transaction identifier sequence generator*.

The *active transactions* contains the list of ongoing transactions that have not committed. This list is updated whenever a transaction starts or commits. It is mainly used for tracking the active transaction with the oldest start timestamp, which is needed for garbage collection. Like [25], the garbage collection of the versions created by a committed transaction is performed after ensuring that there is no older active transaction that could read the versions. The *start-and-commit timestamp sequence generator* is used for issuing start and commit timestamps. Since both timestamps are issued from the same sequence, to find the transactions that ran concurrently with a given transaction T, it is sufficient to choose transactions from the *recently committed* list for which the commit timestamp is greater than the start timestamp of T.

The *transaction identifier sequence generator* assigns a unique identifier to each transaction. This unique identifier plays the role of a temporary commit timestamp for an active transaction. The sequence generator starts from a very large number, larger than the commit timestamp of any transaction that can appear in the lifetime of the system. This timestamp is used in the uncommitted versions written by an active transaction. This approach for timestamp assignment simplifies the distinction between committed and

(a) Impact of the number of concurrent transactions (with $\alpha = 1.4$)



(b) Impact of the percentage of conflicting transactions (with 10 threads)

Figure 6: Trading benchmark experiments

uncommitted versions, because the uncommitted versions are not read by any other active transaction.

Starting a transaction is done by drawing a start timestamp and a transaction identifier from the corresponding sequence generators. Then, a transaction object is created with these values. The transaction object is an encapsulation of the data needed for running, validating and committing a transaction. Besides the start timestamp and the transaction ID, the transaction object keeps track of the undo buffer and the predicate graph. The predicate graph is implemented as a list of root predicates with each predicate storing a list of its child predicates. All operations that interact with the database for reading or writing data require the transaction object, which is passed as a parameter to these operations.

Version Chain Manipulation. All data manipulation operations result in one or more versions. The references to these versions are added to the version chain of the manipulated data object, as well as the undo buffer of the transaction. Each table is implemented as a concurrent cuckoo hash-map [22] of primary keys to data objects. Each data object has an atomic pointer to its version chain called *head*. If multiple uncommitted versions are not allowed for a table (cf. section 2.3.1), the version chain is a simple concurrent lock-free singly linked-list where new versions are added to its *head* or removed from it (in the case of a rollback).

Otherwise, if multiple uncommitted versions are allowed, there are more operations required on the version chain. In this case, a variant of the concurrent Harris linked-list [14] is employed. Each version has an atomic *nextVersion* field. The most significant bit of *nextVersion* is used as *isDeleted* flag. The new versions are always added to the head of the list. If a transaction is rolled back, all *isDeleted* flags of its versions are marked. Otherwise, in a short commit critical section, all its versions are moved next to the previously committed versions in case several uncommitted versions exist (cf. section 2.4.1). The *move* operation is done by marking the version as deleted and creating a duplicate of it and adding it beside the previously committed version if one already exists. The version chain traversal operations (e.g., finding the visible version for a given transaction) are wait-free and responsible for garbage collection of the logically deleted nodes.

Parallel Validation. The list of *recently committed (RC)* transactions is implemented as a concurrent singly linked list. Each committed transaction is added to the head of this list. In the validation phase of a transaction T, the *RC* list is traversed from its head until the last transaction that committed after T started. T is validated against each of these transactions. Next, T obtains a new timestamp (that would be either a commit timestamp in case of success or a new start timestamp otherwise). Then, if *RC.head* is not modified (i.e., no new transaction is committed), validation is done. Otherwise, the same procedure should be repeated from the new *RC.head* to the old one.

6. EVALUATION

We use the TPC-C [31] and TATP [37] benchmarks, as well as some of our own microbenchmarks to show the main pros and cons of MV3C compared to other concurrency control algorithms. We have implemented both MV3C and OMVCC algorithms in C++11. The implementation, excluding the tests and benchmarks, for MV3C is 7 kLOC and that for OMVCC is 5 kLOC. Both implementations use row-based storage and redo logs are stored in memory.

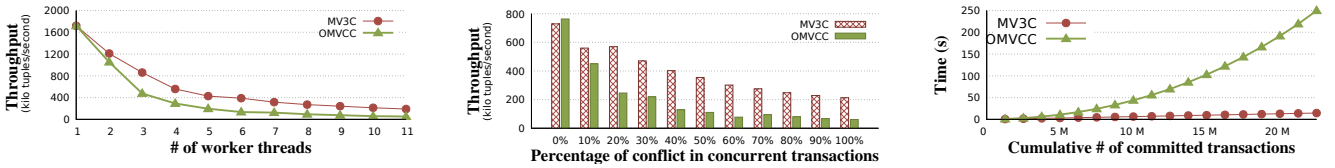
Both MV3C and OMVCC are implemented for single and multi-threaded execution.The results of the multi-threaded experiments are shown in this section. The additional experimental results for the single-threaded execution are shown in Appendix C. All experiments are performed on a dual-socket Intel® Xeon® CPU E5-2680 2.50GHz (12 physical cores) with 30 MB cache, 256 GB RAM, Ubuntu 16.04.3 LTS, and GCC 4.8.4. Hyper-threading, turbo-boost, and frequency scaling are disabled for more stable results. We run all the benchmarks 5 times on a simple in-memory database engine and report the average of the last 3 measurements. In the multi-threaded experiments, each thread is pinned to a physical hardware core. In all experiments, the variance of the results was found to be less than 5% and is omitted from the graphs.

6.1 Rollback vs. Repair

The main advantage of MV3C over OMVCC is repairing a conflicting operation instead of aborting and restarting the transaction. There are different parameters that impact the effectiveness of both MV3C and OMVCC, such as the number and size of concurrent transactions, the percentage of conflicts among concurrent transactions, and the average number of times that a transaction is aborted until it commits. We focus on showing the impact of these parameters on the effectiveness of both MV3C and OMVCC. For this purpose, the TATP and TPC-C benchmarks along with two other mini-benchmarks are used. The first mini-benchmark is based on our Banking example (Example 2). The second one, named *Trading*, is described next.

EXAMPLE 5. **Trading benchmark.** This benchmark is the simplified version of TPC-E [32] benchmark. It simulates a simple trading system, and consists of four tables:

- Security(s_id, symbol, s_price): the table of securities available for trading along with their prices. This table has 100,000 records.
- Customer(c_id, cipher_key): the table of customers along with their encryption/decryption key, used for secure personal data transfer. This table has 100,000 records.
- Trade(t_id, t_encrypted_data): the table for storing the list of trades done in the system. The trade details are stored in encrypted form using the customer's cipher_key, and consists of a timestamp of the trade. This table is initially empty.



(a) Impact of the number of concurrent transactions with 100% conflict ratio

(b) Impact of the conflict ratio with 10 concurrent worker threads

(c) The ripple effect

Figure 7: Banking example experiments

- TradeLine(t_id, tl_id, tl_encrypted_data): the table of items ordered in the trades. The details of each record in the last column are encrypted using the customer's cipher_key and consist of the identifier of an asset, and the traded price. The traded price is negative for a buy order. This table is initially empty.

Moreover, there are two transaction programs in this benchmark: TradeOrder and PriceUpdate. A TradeOrder transaction accepts a customer ID (c_id) and an encrypted payload containing the order details. First, the customer's cipher_key is read. Next, using the customer's cipher_key, the payload is decrypted, which contains a sequence number for t_id, timestamp of the order, and the list of securities along with buy or sell flags for each security. Then, the corresponding securities are read from the Security table so as to get their current prices and then, the respective rows are added to the Trade and TradeLine tables. A PriceUpdate transaction updates the price of a given security from the Security table.

The instances of these two transaction programs conflict, if a security is requested in a TradeOrder, while concurrently a PriceUpdate is updating its price. In order to simulate the different popularities among securities, the sec_id input parameters in both transaction programs are generated following Zipf distribution. △

6.1.1 *Number of concurrent transactions*

Given a fixed percentage of conflicting transactions in a stream of transactions, if we change the number of concurrent transactions, more conflicts occur. The number of concurrent transactions can be controlled by having a fixed number of worker threads for handling a queue of transactions.

Figure 6(a) shows the results of the Trading benchmark (c.f. Example 5) with different number of worker threads. In this experiment, the distribution parameter α for generating sec_ids is 1.4 for both TradeOrder and PriceUpdate transactions. As shown in this figure, MV3C achieves higher throughput compared to OMVCC, as the contention-level increases. MV3C avoids decryption and deserialization of the input data, and re-executes only a small portion of the conflicting TradeOrder transactions, while OMVCC re-executes the conflicting transactions from scratch. Additionally, PriceUpdate consists of a blind write operation, which does not lead to a conflict in MV3C, but creates a conflict in OMVCC.

Figure 7(a) shows the results of a similar experiment for the Banking example (Example 2). In this experiment, there are only TransferMoney transactions, all of which conflict at the end while updating the fee account (line 17 in Figure 2). The figure shows the effectiveness of MV3C compared to OMVCC as the time gap for executing 5 million transactions gets wider for higher contention-levels. This experiment illustrates that even though TransferMoney is a small program, the effect of full re-execution in the case of high contention conflicts is not negligible.

TPC-C benchmark. We implemented the TPC-C benchmark using both MV3C and OMVCC with attribute-level predicate validation. Moreover, we used THEDB [38, 39] for the implementation of TPC-C using OCC [19] and SILO [33] algorithms. Almost all conflicting transactions in TPC-C specification lead to premature abort during execution, instead of reaching the validation phase and failing it. The most number of conflicts happens in TPC-C when running in smaller number of warehouses. Figures 8(a) and 8(b) illustrate the performance of MV3C compared to other algorithms for one and two warehouses respectively. In Figure 8(a), MV3C shows 30% more performance compared to OMVCC with 12 threads. The trend of Figure 8(a) continues with more number of hardware threads. In our benchmark machine, we were limited to 12 physical cores, but running the same experiment in the single-threaded implementation with simulated concurrency shows 2x performance speed-up for 64 concurrent transactions (cf. Appendix C.2). However, the performance speed-up growth decreases by decreasing the contention, and this effect is shown in Figure 8(b). Moreover, the difference between the performance of MV3C and both OCC and SILO stems from the fact that multi-version concurrency control is a better candidate for the TPC-C benchmark.

6.1.2 *Percentage of conflicts among transactions*

For a fixed number of concurrent transactions, the percentage of conflicting transactions determines the success rate between concurrent transactions. If this percentage is 0%, then there are no conflicts. On the other hand, if it is 100%, only one transaction succeeds among all concurrently running transactions; the rest fail, and are repaired or rolled back and restarted depending on the concurrency control algorithm. Consequently, it is expected that MV3C is more effective compared to OMVCC when the percentage of conflicting transactions is higher.

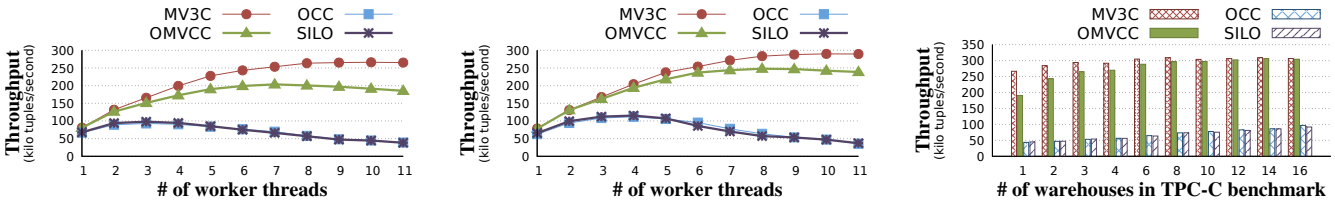
One experiment to illustrate the impact of the percentage of conflicting transactions uses the Banking example. Another program, called NoFeeTransferMoney, is introduced which is similar to TransferMoney, but without the fee payment to the central account. In this experiment, we used 10 worker threads and the percentage of NoFeeTransferMoney programs in the mix is varied. Figure 7(b) shows that MV3C is more effective than OMVCC as the percentage of conflicting transactions increases.

As another experiment, different α parameters of the Zipf distribution of s_id input parameters in Trading benchmark are used with 10 worker threads. The α parameter determines the percentage of conflicting transactions. The results are illustrated in Figure 6(b). This figure demonstrates, once again, that MV3C performs significantly better than OMVCC, as the percentage of conflicting transactions is increased by a larger α parameter.

In addition, as shown in Figure 8, by increasing the contention in the TPC-C benchmark (i.e., lower number of warehouses with a fixed number of worker threads) using MV3C becomes more effective compared to OMVCC, OCC and SILO.

6.2 Overhead of MV3C

Time overhead. Using MV3C as a generic MVCC algorithm is reasonable, only if its overhead in the absence of validation failures or premature aborts is not significant. The conflict-free execution is highly probable in practical low contention scenarios. We consider two approaches for observing this behavior. The first approach ex-



(a) Impact of the number of concurrent transactions in the TPC-C benchmark for warehouse=1

(b) Impact of the number of concurrent transactions in the TPC-C benchmark for warehouse=2

(c) Impact of the conflict ratio in the TPC-C benchmark with 10 concurrent worker threads

Figure 8: TPC-C experiments

ecutes transactions serially, and the second one runs transactions concurrently with no conflicts among them.

As illustrated in Figures 7(a) and 6(a), the overhead of MV3C compared to OMVCC in the serial execution scenario, i.e., with a single worker thread, is under 1%. Figure 7(b) shows that the concurrent execution of transactions with no conflicts has less than 1% overhead using MV3C compared to OMVCC. This small overhead is mostly related to creating the predicate graph in MV3C, instead of creating a list of predicates as in OMVCC. Applying the compiler optimizations for efficiently compiling the closures bound to predicates in MV3C is necessary to achieve such a low overhead.

Furthermore, in the case where the program has a single root predicate and all conflicts occur in this root predicate during the validation phase, the repair phase involves re-executing the whole transaction. Even in this case, the repair phase of MV3C performs similar to a full rollback and restart (as in OMVCC) without additional overhead as shown by our benchmarks.

Memory overhead. In terms of data structures, compared to OMVCC, MV3C has some additional fields in versions and predicates. For each version, MV3C requires an additional pointer field (8 bytes in a x64 system) for maintaining the list of versions produced inside the closure of its parent predicate. As mentioned in section 2.4, this list is used for rolling back the predicates failed in the validation phase. In TPC-C benchmark, the overhead of this extra pointer can be as low as 2% for big records (e.g., Stock table) up to 14% for small records (e.g., History table). The overall overhead of MV3C in TPC-C is 4% extra memory usage.

Moreover, MV3C requires keeping more information for each predicate. The extra fields in a predicate P are used for maintaining the DAG of predicates, the closure assigned to P and the head of the list of versions produced by the closure assigned to P. Accordingly, an MV3C predicate might need twice the memory required for an OMVCC predicate. However, a limited number of predicates are used during the execution of a program, and their memory is reused after the program finishes its execution. For this reason, the memory overhead of predicates is negligible.

7. RELATED WORK

Transaction processing is a fundamental area of database research. Work in this area has resulted in many publications and books [4, 13, 36]. This section covers work done on topics related to MV3C.

7.1 Multi-Version Concurrency Control

Concurrency control techniques from the MVCC family (which includes snapshot isolation) are de facto standards in open source as well as commercial database management and transaction processing systems. PostgreSQL [26], Microsoft SQL Server's Hekaton [7], SAP HANA [29], HyPer [17] and ERMIA [18] are among these systems. In addition, there is recent work proposing efficient MVCC algorithms for in-memory transaction processing [21, 23, 25, 18]. As it was mentioned in section 1, MV3C has an efficient conflict resolution mechanism, which is missing in the existing MVCC algorithms.

For the scenarios that are interesting for MV3C (i.e., high contention data objects and long running transactions), one could propose pessimistic concurrency control algorithms that make use of locking. There is a large body of research on the use of locks in different concurrency control algorithms, and the one used with multi-versioning is Multi-Version concurrency control with Two-Phase Locking (MV2PL) [3, 6]. However, a pessimistic approach to concurrency control yields low performance for long running transactions, as the acquisition of a highly demanded resource by a long running transaction requires either preventing the long-running transaction from committing, or stopping almost all other transactions.

7.2 Partial rollback and restart

The problem of avoiding a complete rollback and restart for a failed transaction has been there since the early days of transaction processing field. Sagas [12] is a pattern for writing long-running transaction programs where transaction programs are divided into smaller actions all of which have a corresponding undo action. Then, actions run sequentially. Each action releases the database resources after it is done. If an action fails, it is enough to roll it back and undo the previous actions until a safe point is reached and then, retry the transaction. This approach is a coarse-grained solution and can be used irrespective of the concurrency control algorithm used inside each action. MV3C is a more fine-grained solution to solve the issue inside one of these actions. Automating the conversion of an arbitrary transaction to Sagas is not straightforward and requires programmer effort, while MV3C provides a mechanism to automatically convert transaction program into MV3C DSL and does not require programmers to provide the undo actions.

Another approach to transaction repair is proposed in [35]. This method requires the transactions to be written in a functional language similar to Datalog. As transaction programs are functional, their impact can be summarized as *delta changes*. Then, in the case of a conflict, it uses the delta changes of committed transactions to fix the conflicted transactions by repeatedly employing the incremental leapfrog triejoin [34] until a fixpoint is reached. This approach mainly focuses on repairing the heavy joins inside the transactions and mostly maps to the case of Figure 1(c) and has no impact on other cases, i.e., Figure 1(a) and (b).

Transaction Healing (TH) [39] is another solution that tries to resolve conflicts among optimistic concurrency control (OCC) transactions at commit time. Despite approaching the same problem, i.e., repairing conflicts between transactions, MV3C and TH differ in several aspects. First, TH works for OCC while MV3C targets MVCC. This choice retains all the differences between OCC and MVCC, one of which is MV3C does not block read-only transactions while TH might block them during validation, healing and commit phases. Second, the validation phase of TH ignores the semantics and predicates of the program and similar to OCC checks

the whole read-set of the program and a big read-set can increase its validation time. However, MV3C considers the read predicates of the program for validation, regardless of the number of read data objects using that predicate. Third, the healing phase in TH is always done in a critical section under the assumption that healing phase is always very short. This is not a valid assumption for a general purpose concurrency control algorithm in which the healing phase might be as expensive as the re-execution of the whole transaction. However, MV3C avoids applying the repair phase in a critical section by default, even though it is still a possible option (cf section 4.3). Fourth, TH is more limited in the type of input transaction programs than MV3C, in that it requires the dependency graph among transaction operations to be statically known. This condition means the approach cannot be used with arbitrary transaction programs with arbitrary control-flow statements inside them. By contrast, even though MV3C uses the available dependency information at transaction compile time, it does not limit the control-flow statements inside transaction programs. MV3C achieves this flexibility by constructing its predicate graph at runtime based on the dependency information encoded in the transaction program, rather than only depending on static dependency information.

7.3 Nested transactions

In the nested transaction model [1, 24, 10], a transaction consists of primitive actions or sub-transactions, which can again be nested transactions. In this model, after detecting a conflict inside a sub-transaction, only the sub-transaction is aborted, the state at its start time is recreated, and it is re-executed. The boundaries of such sub-transactions are user-defined, and checkpointing is the usual technique used for recreating the state.

Irrespective of the operations done in the sub-transaction, checkpointing has to be pessimistic. A checkpoint can be used anywhere within the program and no assumptions about the program can be made during its creation. In addition, the number of extra computation cycles required for checkpointing is not negligible. It requires capturing the whole execution state, and this work is not beneficial to the execution of the main transaction program. Moreover, all work done in a conflicting sub-transaction is lost, and cannot be reused in the subsequent re-execution of the sub-transaction.

MV3C achieves a higher throughput than the generic nested transaction model due to some key differences. MV3C has stricter regulations that do not allow an arbitrary splitting of programs. Instead, the boundaries are defined based on the possible failure points in the program and the dependencies among the operations. Consequently, a faster and more compact checkpointing is achieved, as MV3C tailors the checkpoint for each sub-transaction specifically, unlike the generic checkpointing used in the nested transaction model. Also, MV3C does not commit its so-called sub-transactions, and only tries to commit the transaction as a whole. Thus, MV3C incurs low overhead for executing transactions while still being able to efficiently repair conflicting transactions. This is possible only because of the constraints in the definition of boundaries for the so-called sub-transactions in MV3C.

7.4 Coordination Avoidance

Coordination avoidance (CA) is an approach proposed in [28] and generalized in [2]. In this approach, the semantics of a transaction program is analyzed and the flexibilities in the semantics are used to avoid unnecessarily declaring conflicts between transactions. For example, in a flight reservation system, the number of *remaining seats* in a flight is stored for each flight record. This field is decremented by *flight reservation* procedure calls. Concurrent updates from several *flight reservation* invocations into the *remaining seats* field do not conflict, as long as at least one seat is available and this is the only conflicting operation. This results in schedules that are correct but not necessarily serializable.

In general, even though both MV3C and CA approaches tend to increase the transaction processing performance, they have the following differences: First, unlike CA, MV3C produces serializable schedules. Second, CA requires knowing all the transaction programs in advance in order to analyze their semantics. On the contrary, using MV3C does not limit the system to run a set of predetermined transaction programs, and it can even interoperate with other concurrency control algorithms (cf. section 3). Third, as explained in Appendix B, MV3C has a systematic way of analyzing transaction programs. In spite of that, CA approaches are guidelines for optimizing the execution of transactions at user-level and cannot be fully automated.

7.5 Timestamp allocation

OCC uses timestamps for correctly validating the concurrently executed transactions [19]. The mainstream approach in OCC and its refined variants [19, 27, 5] is to allocate a start timestamp when a transaction starts and a commit timestamp at commit time. These timestamps are assigned by a centralized timestamp allocator. In the multi-version variants of it, both MV3C and OMVCC do a similar job, as explained in section 2.1.

However, as it is shown in [41], a central timestamp allocator can become a bottleneck and limit the scalability of the timestamp-based algorithms. There are several approaches to overcome this bottleneck. Silo [33] tries to solve this issue by allocating coarse-grained (every 40ms) global timestamps. Then, within a timestamp, the transactions are ordered based on their read-after-write dependencies. Another approach taken by TicToc [42] is calculating the commit timestamp of a transaction only based on the timestamp of the data objects read or written by it. This approach not only removes the central timestamp allocation issue, but also creates more opportunities for transactions to commit compared to other OCC approaches. However, unlike MV3C, the resulting schedule of TicToc is not commit order serializable [42]. Moreover, TicToc suffers from phantom anomaly if it wants to perform efficiently [42], while MV3C does not have this issue because of its predicate-based validation mechanism.

Even though both Silo and TicToc try to increase the transaction processing throughput by decreasing unnecessary contention, they still do not have any solution for real contention among transactions. If a transaction fails to validate, they simply abort and restart it from scratch. This not only incurs a performance penalty, but also is unfair against read-mostly transactions [18]. The main issue that MV3C tries to solve is having a low-overhead repair mechanism that is missing from both Silo and TicToc. We believe that the combination of these approaches can lead to new concurrency control techniques that have the best of both worlds.

8. CONCLUSIONS

In this paper, we proposed MV3C, a conflict resolution mechanism for optimistic MVCC, proved its correctness and showed its effectiveness in various scenarios. Depending on the scenario, MV3C can achieve more than an order of magnitude improvement in transaction processing throughput, while in the worst-case does not have any significant overhead compared to optimistic MVCC.

Acknowledgements

The authors wish to thank Phil Bernstein and Ashkan Norouzi-Fard for many helpful suggestions and comments on this paper. This work was supported by ERC grant 279804.

9. REFERENCES

[1] J. Aspnes, A. Fekete, N. Lynch, M. Merritt, and W. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Fourteenth International Conference on Very Large Databases*, pages 431–444, Aug. 1988.

[2] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, Nov. 2014.

[3] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Trans. Database Syst.*, 5(2):139–156, June 1980.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] M. J. Carey. Improving the performance of an optimistic concurrency control algorithm through timestamps and versions. *IEEE Trans. Softw. Eng.*, 13(6):746–751, June 1987.

[6] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, SIGMOD '82, pages 184–191, New York, NY, USA, 1982. ACM.

[7] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, 2013.

[8] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 7(4):277–288, Dec. 2013.

[9] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.

[10] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. *J. Comput. Syst. Sci.*, 41(1):65–156, Aug. 1990.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[12] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, Dec. 1987.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[14] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[15] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 international conference on Management of data, SIGMOD '10*, pages 603–614, New York, NY, USA, 2010. ACM.

[16] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locks. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 143–147, New York, NY, USA, 1981. ACM.

[17] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[18] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1675–1687, New York, NY, USA, 2016. ACM.

[19] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[20] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4), 1964.

[21] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, Dec. 2011.

[22] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 27:1–27:14, New York, NY, USA, 2014. ACM.

[23] T. Merrifield and J. Eriksson. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 127–139, New York, NY, USA, 2013. ACM.

[24] J. E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[25] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.

[26] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012.

[27] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 81–88, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[28] A. Reuter. Concurrency on high-traffic data elements. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '82, pages 83–92, New York, NY, USA, 1982. ACM.

[29] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. SIGMOD '12, pages 731–742, New York, NY, USA, 2012. ACM.

[30] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[31] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11. http://www.tpc.org/tpcc.

[32] Transaction Processing Performance Council. TPC-E Benchmark Revision 1.13.0. http://www.tpc.org/tpce/.

[33] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[34] T. L. Veldhuizen. Incremental maintenance for leapfrog triejoin. *CoRR*, abs/1303.5313, 2013.

[35] T. L. Veldhuizen. Transaction repair: Full serializability without locks. *CoRR*, abs/1403.5645, 2014.

[36] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[37] A. Wolski. TATP Benchmark Description (Version 1.0). http://tatpbenchmark.sourceforge.net, Mar. 2009.

[38] Y. Wu. THEDB (Cavalia). https://github.com/Cavalia/Cavalia, 2016.

[39] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1689–1704, New York, NY, USA, 2016. ACM.

[40] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment*, 9(5):444–455, 2016.

[41] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[42] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, 2016. ACM.

APPENDIX

A. SERIALIZABILITY PROOF

Before proving the serializability of MV3C, Lemma 2.4 is proven.

Proof of lemma 2.4. Assume that the database is duplicated at timestamp S' into two identical instances, D_1 and D_2. On instance D_1, T aborts and restarts using the new start timestamp S', resulting in the predicate graph $G''(S')$. On instance D_2, the Repair algorithm is applied on $(G(S), S')$, which results in the predicate graph $G'(S')$. It should be noted that both $G'(S')$ and $G''(S')$ have the same root predicates, without considering their lists of versions. The reason is that the creation of a predicate depends on its parent predicate. The root predicates do not have a parent by definition, so they are created regardless of the database state. It is also guaranteed that they are also not removed from the predicate graph by the Repair algorithm, as they are not descendants of any other nodes. Moreover, in line 3 of the Repair algorithm, the results of the Validation algorithm on $(G(S), S')$ are used, i.e., L_1 and L_2. Let X be an arbitrary node selected from $G(S)$. There are three cases.

Case 1: X is in L_1. Then, X is a valid predicate, and based on Lemma 2.1, all of its ancestors are valid, too. Let Y be the list of root predicates that are the ancestors of X. Nodes in Y are common in both $G'(S')$ and $G''(S')$. Hence, as part of executing T on D_1, the corresponding nodes in $G''(S')$ are created and the closures bound to them are executed. As nodes in Y are valid nodes, they read the same data as S, and take the same program flow, which results in creating X with the same list of versions inside it.

Case 2: X is in L_2 and has no incoming edge from another node in L_2. Then, either X has no parent, or its parents are in L_1. In both cases, X is created while executing T from scratch on D_1. If X has no parent, then it exists regardless of executing under timestamp S or S'. If X has a list of parents Y in L_1, then based on Case 1, all nodes Y exist in $G''(S')$ and they are valid. As the execution of their closures using the timestamp S' is the same as S, they create the same child predicates and X is among them. Lines 5-14 of the Repair algorithm prune X and re-execute it. Then, the closure bound to X is re-executed in line 13 of the Repair algorithm using timestamp S', which is identical to executing it on D_1. And, it results in the same descendant nodes and list of versions for X in both $G'(S')$ and $G''(S')$.

Case 3: X is in L_2 and has an incoming edge from another node in L_2. In this case, X has an ancestor, Z, from L_2 that falls into Case 2. Thus, Z is pruned and X is removed from $G(S)$ in lines 7 and 10 of the Repair algorithm. As it is shown in Case 2, the same descendant nodes are recreated for Z in both $G'(S')$ and $G''(S')$.

Thus, $G'(S')$ and $G''(S')$ are equivalent. □

Next, the serializability of MV3C that is claimed in Theorem 2.1 in section 2.6 is proven.

Proof of theorem 2.1. Based on Definition 2.3, the versions created by a transaction become visible to the other transactions only after the transaction commits. Accordingly, the operations done by uncommitted transactions cannot affect the serializability of the committed transactions. Thus, the uncommitted transactions can safely be ignored, and only the committed transactions are considered in this proof.

By showing that all the dependencies between different transactions executed under MV3C have the same order as their commit timestamp, it can be proven that any execution of transactions under MV3C is serializable in commit order. Read-only transactions read all committed versions that are committed prior to their start. Moreover, the commit timestamp of a read-only transaction is the same as its start timestamp, as if it is executed at that point in time.

An update transaction starts, executes, and then enters the validation and commit phase. If the validation is successful, a commit timestamp C is assigned to the transaction. Otherwise, the transaction acquires a new start timestamp S' and enters the repair phase. As it is already shown in Lemma 2.4, the predicate graph resulting from applying the Repair algorithm at S' is the same as the one resulting from aborting and restarting the transaction from scratch at timestamp S'. Then, the repaired transaction enters the validation phase again, and this cycle continues until the transaction succeeds in the validation phase.

Thus, it is sufficient to prove that for an update transaction that starts with timestamp S, executes, successfully passes the validation phase and commits, the visible effect of the transaction is as if it is executed at one point in time, C, where $S < C$. The proof is done by contradiction, and is similar to the serializability proof in [25]. There are some modifications for allowing write-write conflicts in MV3C, as this kind of conflicts leads to premature abort and restart in [25].

Let T_1 be an update transaction from the committed projection of H with start timestamp S_1 and commit timestamp C_1. Suppose that the execution of T_1 cannot be delayed until C_1. Then, there is an operation, $o1$ in T_1 that conflicts with an operation $o2$ in another transaction, T_2 (started at S_2 and committed at C_2), with $o1 < o2$ and T_2 committed during the lifetime of T_1, i.e., $C_2 < C_1$. There are four cases corresponding to the possible combinations of two operations $o1$ and $o2$.

Case 1: both are reads. In this case, $o1$ and $o2$ can be swapped and this contradicts the assumption that $o1$ and $o2$ are conflicting.

Case 2: $o1$ is write and $o2$ is read. Based on Definition 2.3, the version written by $o1$ is not visible to $o2$, as $S_2 < C_1$. Thus, it contradicts the assumption that $o1$ and $o2$ are conflicting.

Case 3: $o1$ is read and $o2$ is write. By C_1, the version V_2 written by $o2$ is committed and exists in the undo buffer of T_2, as $C_2 < C_1$. The operation $o1$ is done via a predicate P_1. Hence, while validating T_1 at C_1, the Validation algorithm matches P_1 against V_2. If P_1 matches, T_1 fails validation and cannot commit at C_1. This contradicts the fact that T_1 with this commit timestamp is in the committed projection of H. Otherwise, if P_1 does not match V_2, then it contradicts the assumption that $o1$ and $o2$ are conflicting.

Case 4: both are writes. In this case, $o1$ and $o2$ can be swapped and this contradicts the assumption that $o1$ and $o2$ are conflicting. Basically, an individual write operation acts as a blind-write. Thus, the blind-write operation in $o1$ can be delayed until C_1. If a write operation is not a blind-write, there is a read operation done before it, and this case is converted to either Case 2 or Case 3. □

B. TRANSLATING INTO MV3C DSL

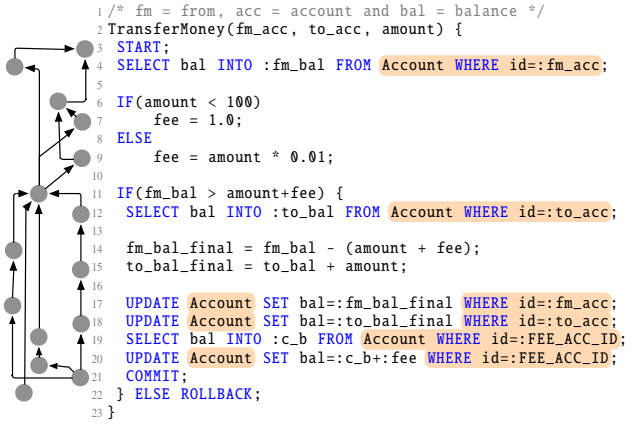
In order to execute a transaction under MV3C, the transaction program can also be given in the form of a PL/SQL-like language. In this case, the program goes through a translation pipeline in order to be converted into the MV3C DSL described in section 2.2. There are three stages in the compilation pipeline, described next.

First stage. In this stage, the representation of the program is converted into a *dependency graph*. The dependency graph, also known as program dependence graph representation [11], is a graph where nodes are program operations, and an edge from node A to

node B means the operation in node A depends on the results of the operation in node B in order to execute.

The operations provided in the PL/SQL-like language can be categorized into read, computation, data manipulation, or a combination of these types. As MV3C DSL does not have combined operations, the combined operations are decomposed into primitive ones during the construction of dependency graph. This decomposition is done in such a way that preserves the dependencies. It should be noted that read operations in the graph contain the data selection criteria that is represented by a predicate in MV3C DSL. Consequently, all the read operations in the graph are marked as predicate nodes.

The dependency graph is just another representation for the actual program with more data locality. In the dependency graph, the operations for reading data come closer to the operations for manipulating and consuming the data. In addition, the marked predicate nodes in the graph form an overlay graph built on top of the dependency graph. The *predicate overlay graph* is a DAG where nodes are predicates and there is an edge from node p1 to node p2 if there is a path in the *dependency graph* from p1 to p2. As stated earlier in section 2, the versions created by an uncommitted transaction are hidden from other transactions. Thus, the only reason for a transaction T to fail in the validation phase is reading some outdated data objects, which were committed during the execution of T. Therefore, the predicate overlay graph can be viewed as the possible failure points in the program.



```
1 /* fm = from, acc = account and bal = balance */
2 TransferMoney(fm_acc, to_acc, amount) {
3  START;
4  SELECT bal INTO :fm_bal FROM Account WHERE id=:fm_acc;
5
6  IF(amount < 100)
7      fee = 1.0;
8  ELSE
9      fee = amount * 0.01;
10
11 IF(fm_bal > amount+fee) {
12  SELECT bal INTO :to_bal FROM Account WHERE id=:to_acc;
13
14  fm_bal_final = fm_bal - (amount + fee);
15  to_bal_final = to_bal + amount;
16
17  UPDATE Account SET bal=:fm_bal_final WHERE id=:fm_acc;
18  UPDATE Account SET bal=:to_bal_final WHERE id=:to_acc;
19  SELECT bal INTO :c_b FROM Account WHERE id=:FEE_ACC_ID;
20  UPDATE Account SET bal=:c_b+:fee WHERE id=:FEE_ACC_ID;
21  COMMIT;
22 } ELSE ROLLBACK;
23 }
```

Figure 9: An example transaction program in a PL/SQL-like language with its dependency graph.

Example 3 (continued). The result of applying the first stage on the transaction program shown in Figure 2 is illustrated in Figure 9. The dependency graph of the program is shown on the left-side of the Figure 9, where the operation on each line is represented by a gray circle, and arrows between circles show the dependency among the operations. In addition, the combined operation (read and write) on line 17 of Figure 2 is converted into two primitive operations (one for read and one for write) on lines 19 and 20 of Figure 9. △

Second stage. In the second stage of translation into MV3C DSL, the *dependency graph* is partitioned into sub-graphs satisfying the following properties:

- Each sub-graph has to have at least one predicate node inside it, as it is the representative of an operation that can fail validation. The predicate nodes form the roots of the sub-graphs.
- For all nodes in the sub-graph, every node that depends on it, is also in the sub-graph. This property ensures that the sub-graph represents the minimal set of operations that have to be re-executed if the predicate fails.

A *failure unit* is the smallest sub-graph (with respect to the number of vertices) satisfying the above properties. The translation process aims at partitioning the dependency graph into *failure units* in a bottom-up manner.

The above procedure is repeated recursively after collapsing the *failure units* that are already found into black-box nodes (i.e., the predicates inside these failure units are not considered anymore), until predicates appear only as roots. The output of this process is the nested partitions of the graph, where each partition can fail the validation phase. If there are other predicates depending on the root predicate nodes, they are in a nested sub-graphs.

Example 3 (continued). After applying stage 2 on the dependency graph shown in Figure 9, two failure units are found initially. These two failure units are illustrated in Figure 3 using dark gray color, which are represented by predicates P_2 and P_3. Then, these two failure units are collapsed into two black-box nodes, and stage 2 runs recursively and in the next round, another failure unit is found. This failure unit is shown in Figure 3 using light gray color, which is represented by predicate P_1. △

Third stage. In the final stage of translation, the transaction program is generated using the dependency graph that is partitioned into *failure units*. The generated program has the following characteristics.

- It is equivalent to the input program, as the program logic should remain intact.
- Corresponding to each *failure unit* in the dependency graph, there is a part of the generated program that contains the responsible predicates. This part of the program depends on the results of those predicates.
- Each data manipulation operation is given the references to its direct parent predicates. These references are required for cleaning up the conflicted state, in the case of a conflict happening in the program represented by this *failure unit*.
- It has to build the predicate dependency graph, when the generated program is executed.

It should be noted that the predicate overlay graph in the previous stage is not necessarily equivalent to the runtime predicate dependency graph, even though those are similar. The difference is that the runtime predicate dependency graph has the actual values for predicate parameters while compile-time predicate dependency graph contains symbols for them. In addition, a predicate (and its assigned closure) can be instantiated several times at runtime, e.g., when it is evaluated in a loop construct. The program generated using each *failure unit* is named a *closure transition*, as it encloses all the operations that depend on one or more predicates, along with the context variables required for executing it.

The algorithm for generating the transaction program using the partitioned dependency graph starts with the top-level *failure units*. By definition, there is no overlap between the *failure units* and therefore, the generated code for those do not overlap either.The code generation involves the following three steps :

1. The root predicates are extracted from a *failure unit*.
2. The code for instantiating the root predicates is generated.
3. The code for the nodes dependent on the root predicates is generated and wrapped inside a closure. This closure is passed to the *execute* function of the root predicate. The dependent

nodes are either computational nodes, data manipulation nodes or nested *failure units*.

- Generating code for computational nodes, which does the intended computation, is straightforward.
- For data manipulation nodes containing insert, update or delete operations, the parent predicates have to be correctly registered in the generated code.
- For each nested *failure unit*, the same procedure is repeated recursively. The only difference is that, while generating code for predicates in the nested *failure units*, their direct parent predicates have to be indicated. Moreover, the returned results of the parent predicates can also be used inside the operations of the nested *failure units*.

C. EXTENDED EVALUATION

A common approach to evaluate a concurrency control technique is having a single thread of execution while concurrency among transactions is simulated. The same approach is taken in [25] while evaluating OMVCC. This technique leads to more deterministic results, avoids the overheads of concurrent execution and focuses on showing the impact of the concurrency control algorithm. Moreover, the number of concurrent transactions is not bound to the number of available CPU cores on the machine used for experiments. The concurrency among transactions is realized in these implementations by dividing programs into smaller pieces and interleaving these pieces with ones from other programs. Then, at each step, a single transaction piece gets executed.

A program can be divided into at least three pieces: (1) the start transaction command, (2) the logic of the transaction program itself, and (3) the commit transaction command. In addition, the logic of the transaction program can be further divided into smaller pieces. However, for MV3C transactions, the concurrency level only depends on the number of active transactions (i.e., the transactions that started but are not committed) rather than the number of interleaved program pieces, because transactions only see the committed state. In our experiments, unless otherwise stated, we use the notion of *window* to control the number of concurrent transactions. Given a window size N, which implies having N concurrent transactions, N transactions are picked from the input stream and all of them start, then they execute, and finally they try to validate and commit one after the other. Here, the transactions that fail during the execution are rolled back and moved to the next window. Transactions that fail validation acquire a new timestamp immediately and their executions are moved to the next window. If $N = 1$, then the execution is serial.

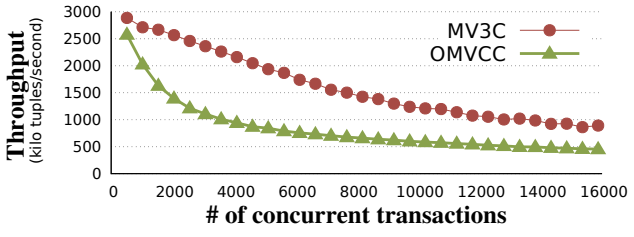


Figure 10: TATP experiment with scale factor 1

C.1 TATP Benchmark

We have implemented the TATP benchmark in both MV3C and OMVCC. Figure 10 shows the results of executing 10 million TATP transactions with different window sizes. This experiment is done for scale factor 1 with non-uniform key distribution and attribute-level validation. The input data is generated using OLTPBench [8]. Once again, this experiment confirms the increased effectiveness of MV3C compared to OMVCC, as the window size increases. Note that even with non-uniform key distribution, the number of conflicting transactions for small window sizes is low, as 80% of the workload consists of read-only transactions. Thus, MV3C and OMVCC show similar results for small window sizes; the difference can only be seen in bigger windows. Moreover, as transaction programs in the TATP benchmark are very small, the main advantage of MV3C over OMVCC is that the former optionally accepts write-write conflicts, while the latter prematurely aborts after a write-write conflict. This decision leads to having no conflicts among Update_Location transaction instances in MV3C, while it leads to aborts in OMVCC.

C.2 TPC-C Benchmark

In addition to the multi-threaded experiments for TPC-C, we conducted further experiments with simulated concurrency among transactions. In the simulated concurrency, we are not limited to the number of physical cores available in the system. Figure 11 illustrates the results of running TPC-C benchmark up to window size 64. The results shown in Figure 11 are interesting, not only because it confirms the effectiveness of MV3C compared to OMVCC, as the number of concurrent transactions increases, but also because its relative difference compared to OMVCC matches the multi-threaded results shown in Figure 8(a).

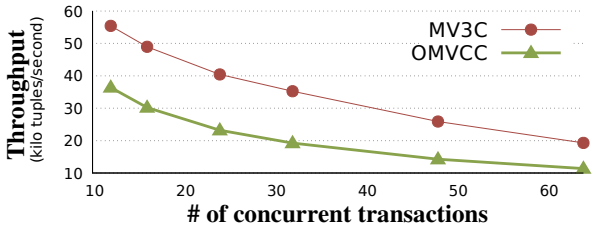


Figure 11: TPC-C experiment with 1 warehouse

C.3 The Ripple Effect

The time saved by repairing the transactions instead of aborting and restarting them from scratch can be used to reduce the load on the system. To show this effect, we ran an experiment where two streams issue transactions at a constant rate. The first stream issues its transactions at almost the serial transaction processing rate, while the second issues its transactions at a much slower pace.

Figure 7(c) shows the results of one such experiment. In this experiment, the TransferMoney transaction program from the Banking example (Example 2) is used in both streams. The schedule is generated using logical time units. The time taken for executing one TransferMoney is assumed to be 250 units for both algorithms. We choose three quarters of this time, i.e., 187 units as the time for partially re-executing conflicting blocks in MV3C, based on the results of Figure 7(a). The two streams produce TransferMoney transactions at the rate of 251 units and 72,000,000 units respectively. This figure shows not only that the time for processing MV3C transactions is lower, but also that the overall behavior of MV3C is completely different than OMVCC. Basically, this experiment shows that a longer conflict resolution approach not only affects an individual transaction, but also has a compound effect on subsequent transactions. The reason is an increase in the probability of having many concurrent transactions, which results in even more conflicts.