Generic Version Control: Configurable Versioning for Application-Specific Requirements

[Vision Paper]

Gunce Su Yilmaz
Saarland University
Saarland Informatics Campus
Germany

Jens Dittrich
Saarland University
Saarland Informatics Campus
Germany

ABSTRACT

Even though Multi-Version Concurrency Control (MVCC) and Git look very different from a user point of view, both systems conceptually do very similar things. In this paper, we thoroughly compare both systems w.r.t. their logical and physical differences and commonalities. We argue that both systems converge to a common one-size-fits-all system. One key to that system is the observation that nested transactions and the Git commit-graph are conceptually the same thing. Another crucial insight is the need for database researchers to rethink conflict resolution and reconciliation. This shift can reduce abort rates, address concurrency issues within the database layer, and eliminate unnecessary round-trips to the application layer. Based on our observations, we propose a unified system, *Generic Version Control (GenericVC)*, combining the best of both worlds. In fact, by combining features from both Git and MVCC, we obtain more than the sum of its parts: the ability to support new hybrid use cases.

1 INTRODUCTION

Alice interacts with an information system. She inspects some data and changes them, keeping all her changes in 'draft mode', thereby creating her own private version of the data. Then she inspects some more data and changes them as well, adding more data to her draft. Eventually, she decides to submit her changes. Now, her changes become part of the 'official' version of the data in that information system.

This first paragraph was a bit imprecise as we did not specify what we actually mean by the term "information system". Let's assume that the information system Alice is working with is a Git repository. Dear reader, please read the first paragraph again.

Done? Now let's assume that by the term "information system" we mean a DBMS using MVCC for concurrency control. Dear reader, please read the first paragraph again.

The paragraph worked in both cases? But why? We are talking about two very different systems! In the case of Git, Alice creates a feature branch, adds commits to that feature branch, and eventually merges (or rebases) her feature branch onto the main branch. In the case of a DBMS, Alice starts a transaction and performs changes through SQL statements that are part of that transaction, implicitly creating her own version of the data through MVCC. Eventually, she commits her transaction which lets her changes to become the official version of the database. In both cases, Alice does something that is *conceptually* the same thing. At the same time, in both cases, Alice uses totally different systems, algorithms, and implementations for the same concepts. Why is that?

We argue that the historical division between versioning systems (like Git) and versioned database stores (like in MVCC implementations) can be bridged, given the similarities between these two approaches. To explore this hypothesis, we present GenericVC, a unified system that can serve both Git-style and MVCC-style workloads. While we recognize the value of having specialized systems, our research suggests that a generalized approach may offer benefits such as reduced maintenance overhead, simplified branch management for databases with branching capabilities, and an improved conflict resolution mechanism. In contrast to prior work on branched databases [8, 12, 17, 18], we identify Git-style branching and database transactions as different implementations of a single concept. In fact, GenericVC can handle database transactions and Git-style branches in a single transaction layer using nested transactions. Our approach also has implications for existing MVCC-only systems. GenericVC rethinks conflict resolution in MVCC by incorporating concepts from Git, such as configurable conflict detection and reconciliation, which enables new hybrid use cases that are not possible to handle with existing Git-only or MVCC-only systems.

In summary, this paper makes the following contributions:

(1) We provide a thorough and structured logical and physical comparison of MVCC and Git, offering insights into their versioning, storage, transaction processing, and garbage collection mechanisms, along with background and related work (Sections 2 & 3).
(2) We propose a generalized and configurable system (GenericVC) that simultaneously accommodates Git-style version control and MVCC-style concurrency control, unifying features of both worlds into a common system (Section 5).
(3) We propose two key functions (`detect_conflicts` and `reconcile`) to be invoked in the commit validation phase of a transaction, which MVCC systems can expose to developers to support configurable conflict detection and reconciliation for application-specific use cases (Section 5).
(4) We identify hybrid use cases enabled by GenericVC, showcasing its versatility over systems with rigid rules and architectures. Our system supports a wide range of applications within a single system and is not merely another relational database with branching capabilities (Section 6).

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, 2025, Amsterdam, The Netherlands.

2 LOGICAL COMPARISON OF MVCC AND GIT

MVCC and Git [7] share significant conceptual and functional similarities. We have analyzed the similarities and differences between the two systems and organized our findings under two categories: *logical comparison* (this section), which focuses on the high-level concepts and functionalities, and *physical comparison*, which delves into how the data store is organized and accessed (Section 3).

Terminology. We first need to clarify what we mean by the term 'version.' MVCC tracks changes to a *tuple* by creating a new physical copy of that tuple for every update. That tuple also contains (hidden) metadata fields like *begin* and *end* timestamps. Hence, a version in the MVCC context is a timestamped physical copy of the same logical tuple in different states. In contrast, Git tracks changes to a *document* by creating a new physical copy for every update. The document is stored in a file whose relative path to Git's object store is the hash of the document's content. Therefore, a version in the Git context is a hash-addressable physical copy of the same logical document in different states. Notice that we deliberately use the term 'document' instead of 'file' to prevent confusion between the file as the fundamental unit of storage in a file system and the content of a file, i.e. the byte-stream, that is versioned in Git.

Transactions. A transaction in a relational DBMS is an indivisible unit of work consisting of one or more database operations, conforming to the ACID properties. In MVCC, each transaction creates versions only visible to itself until the transaction commits, ensuring that intermediate states are hidden from concurrent transactions (assuming isolation level of read committed or above). In Git, creating a feature branch, editing documents, and merging the branch into the main branch is similar to a database transaction. A branch in Git is a pointer to a commit object, which represents a snapshot of the repository at a specific point in time (commit objects explained in more detail in Section 3). When a branch is created, it initially points to a commit object. As changes are committed, the branch pointer moves to new commits, forming a linear commit history akin to multiple database operations within a transaction. This linear chain of commits is also referred to as a branch. To differentiate between the branch as a pointer and the logical branch representing the history of commits, we will refer to the latter as a *feature branch*. Commits in a feature branch are like a transaction's private space, where modified document versions are invisible to the main branch until merged. However, transactions in Git do not fully conform to the ACID properties, as discussed next.

Atomicity. When a transaction is committed in MVCC, new tuple versions become publicly visible, and older versions become invisible to subsequent transactions. Similarly, in Git, the *merge* command makes changes on the feature branch visible to the main branch. A successful merge (usually the 3-way merge that merges two branched versions using a third reference base version) creates a new commit object and updates the main branch pointer to point to this commit. If a transaction aborts in MVCC, its versions remain invisible and are later garbage collected. When a merge operation fails in Git, Git does not create a commit object and leaves a dirty version of the conflicted files with rows from both versions being merged for manual handling. Since the commit object is not created, none of the changes are visible to main branch or any subsequent branches. By employing this all-or-nothing behavior, Git ensures atomicity.

Consistency. MVCC systems provide strong consistency by enforcing integrity constraints for all writers, ensuring that a transaction moves the database from one valid state to another. Git offers weaker consistency guarantees. Git has a mechanism called *hooks* to enforce constraints, such as successful compilation, to which all merges must adhere, but defining these constraints is the responsibility of the developer. Git also provides some consistency through its conflict resolution strategy. When two feature branches concurrently update a document, Git detects conflicts if changes affect not only the same line but also the adjacent lines. The idea behind this strategy is that concurrent changes to adjacent lines likely cause syntactical or semantic issues and require manual handling. Git enforces this strategy in its 3-way merge algorithm by requiring context lines (unchanged lines) around modified lines for a successful merge.

Isolation and the Commit Validation Phase. MVCC systems offer isolation levels by implementing access rules and conflict resolution mechanisms. Most MVCC implementations use metadata fields, like *begin* and *end* timestamps, to set access rules to avoid dirty reads, non-repeatable reads, and phantom reads. They also prevent the lost update anomaly by using a first-writer-wins rule [10, 14, 15], ensuring isolation levels up to repeatable read and snapshot. To achieve serializable isolation, read-write conflicts are resolved during commit validation, using methods like dependency graphs [2, 19], read and scan sets [5, 10], or predicate logging [14].

Git offers guarantees similar to snapshot isolation. A feature branch views the repository's snapshot taken at the time of its creation, avoiding dirty reads, non-repeatable reads, and phantoms, unless the developer deliberately merges changes from other feature branches. However, not all lost update problems are avoided due to Git's unique write-write conflict resolution policy. For instance, if two developers branch from the same commit and edit the same line of a document with exactly the same text, Git will not detect a conflict during the merge. In MVCC, if two transactions concurrently add 1 to an integer field with an initial value of 5, a Git-like merge would result in field being set to 6 and not 7. This behavior is acceptable in Git but unsuitable for most MVCC use cases that deal with numeric attributes. Git also does not offer serializable isolation due to its less strict commit validation phase (merge). Whereas MVCC checks for read-write conflicts during commit validation, Git only checks for write-write conflicts, making it prone to write-skew anomalies.

Durability. DBMSs ensure durability in the event of a crash by writing changes to a write-ahead log before writing to the database. Git does not ensure such durability but provides some data integrity in the event of a crash. Merging a feature branch into the main branch involves finding a merge base, merging histories into a tree object, creating a commit object, and updating the main branch pointer. If a crash occurs during this process, Git does not keep logs to repeat the process. However, the crash is unlikely to corrupt the repository, as changes are not visible to the main branch until the last step, where the main branch pointer is updated.

3 PHYSICAL COMPARISON OF MVCC AND GIT

Physical Storage Layout. In a row-based database layout, versions of multiple tuples are packed together within a page, assuming the maximum tuple size is smaller than the page size. Conversely, Git stores each document version as a blob (binary large object), which may span multiple pages. Besides blobs, Git stores tree and commit objects. Tree objects, structured as directories, point to blobs created since the last commit. A commit object points to the top-level tree object and to parent commit objects, representing a snapshot of the entire repository at a specific point in time.

Version Storage Schemes and Version Manipulation. Wu et al. categorize version storage schemes in MVCC implementations into three types: append-only, time-travel, and delta storage [20]. In append-only storage, each update to a tuple creates a new version in the same table. In time-travel storage, the latest versions of tuples are stored in a main table, with older versions in a separate table. Delta storage updates a tuple in place and stores a delta, containing the prior values of modified attributes, in a separate storage. Git has two version storage schemes: loose and packed objects. When a document is modified, Git saves the **entire** document as a new blob, not just the modified parts, called a loose object. Loose object format is similar to append-only and time-travel schemes. When the number of loose objects grows large or when the user runs Git's garbage collection command, Git packs objects into a single binary file. In this packed format, Git retains the most recent versions of the documents in their entirety and stores reverse deltas for previous versions, similar to delta storage in MVCC.

Version Chains. In MVCC, versions of a tuple are linked together via pointers, forming a version chain. The index entry points to one of these versions, either the oldest or the newest depending on the ordering [20], and the DBMS traverses the chain to find the desired version. Git stores versions in a commit-graph, a content-addressed directed acyclic graph that maintains all the changes in a repository. A commit object in the graph points to a parent commit or commits. The commit object also points to a tree object, which itself points to other tree and blob objects, representing the new versions of the updated documents in this commit. Hence, in Git, the document versions (blobs) do not directly point to each other, and Git needs to traverse the commit-graph by following the parent pointers of the commit objects to find an older version of a document.

Version Access. To access a tuple version, MVCC performs tuple visibility checks using metadata fields like *begin* and *end* timestamps. A transaction can view a version of a tuple if its *begin* timestamp falls within the version's timestamps, allowing the transaction to query a consistent snapshot of the data. A DBMS can also support time-travel functionality by retaining old tuple versions in the store or leveraging the (unpruned) log file to reconstruct old versions. This allows transactions to access older versions using the 'AS OF' sub-clause. To access the current version of a document, Git traverses the commit-graph, following parent pointers of commit objects to find the commit, where the document was last modified. Git then fetches the blob object that represents the document in this commit. To access the older versions of a document, the developer can use the content hash of the document or check out the commit object that points to the older version, prompting Git to further traverse the commit-graph to find the corresponding commit object.

Garbage Collection. Long version chains in MVCC can cause performance and storage overhead, so a critical function of the garbage collection protocol is to detect and remove expired (as well as aborted) versions. Garbage collection in a DBMS usually involves detecting versions no longer visible to any active transactions, removing references to these versions from version chains (unlinking), and reclaiming the memory space [14]. Garbage collection command in Git also aims to reduce storage overhead but retains all versions. Instead of removing old versions, Git packs loose objects into the packed file format, keeping the most recent version and reverse deltas for previous versions. This transforms Git's append-only storage into delta storage. In contrast to MVCC, Git does not garbage collect versions created by a failed merge operation, leaving them in the working directory for manual conflict resolution.

4 OTHER RELATED WORK

This section describes prior work on bringing Git-style versioning to the database world. Multiple studies implement branching logic on top of existing relational data storage systems. Decibel [12] and TardisDB [18] use bitmap-based storage techniques to do so, where each bit in the bitmap represents an active tuple in the corresponding branch. OrpheusDB [8] and MusaeusDB [17] extend data tables with a metadata field called record identifier (rid) and maintain versioning tables to map branches to rids. While these studies provide valuable insights into the design of database systems that support branching, they primarily focus on extending the capabilities of existing database systems with an extra branching layer. In contrast, our goal is to identify branching and database transactions as two different implementations of a single concept. This insight enables GenericVC: we handle transactions and branches through the same techniques and allow for application-specific conflict detection and reconciliation strategies. Finally, MindPalace [16] is a versioned data system that introduces the concept of "auto-mergeability" and implements a strategy for reconciling conflicts by merging two modification histories into a single, combined history that contains all modifications from both histories. GenericVC provides a conflict reconciliation function that can be customized to meet application-specific requirements. This approach allows for semantic reconciliation as well, which was deliberately left out of "auto-mergeability" definition in MindPalace.

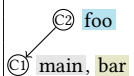
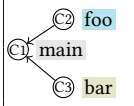
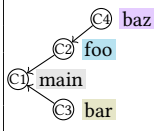
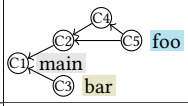
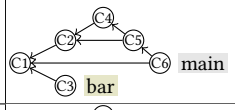
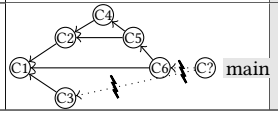
5 GENERICVC: CONFIGURABLE VERSIONING

Our comparative analysis of MVCC and Git highlighted that both systems have overlapping concepts and mechanisms with subtle differences to support their respective use cases. In the following, we propose three logical dimensions to describe these overlaps and generalize them into our unified common system GenericVC: (1.) Git Commit Graph as Nested Transactions, (2.) Isolation Levels, and (3.) Conflict Prevention, Detection, and Reconciliation.

5.1 Git Commit Graph as Nested Transactions

As discussed in Section 2, a feature branch in Git is very similar to a database transaction. Both a feature branch in Git and a transaction in MVCC operate in their private spaces, ensuring that intermediate states of the data remain hidden from other concurrent transactions.

Table 1: Branching in Git vs MVCC with nested transactions. Work on different branches is color coded. Circles in the commit graph represent commit objects (tree and blob objects are omitted for simplicity). Arrows represent parent-child relationships among commit objects.

	Git	Explanation		MVCC
		Version Graph	Comments	
1	git init	C1 main	Initialize main branch.	MVCC main;
2	git branch foo	C1 main, foo	Create a feature branch named foo.	foo = main.begin_transaction();
3	git branch bar	C1 main, foo, bar	Create another feature branch named bar.	bar = main.begin_transaction();
4	git switch foo echo 'a=42' » doc.txt git commit -am 'fixes bug 12'		Switch back to branch foo. Create new document version V1. Make change visible on branch foo and any future branch forking from foo. P.S. V1 is not visible to main and bar yet.	foo.update('doc.txt', '…a=42\n ');
5	git switch bar echo 'a=43' » doc.txt git commit -am 'fixes bug 12'		Switch back to branch bar. Create new document version V2. Make change visible on branch bar and any future branch forking from bar. P.S. V2 is not visible to main and foo yet.	bar.update('doc.txt', '…a=43\n ');
6	git switch foo; git switch -c baz echo 'a=44' » doc.txt git commit -am 'fixes bug 15'		Create a feature branch named baz from foo. Create new document version V3. Make change visible on branch baz and any future branch forking from baz. Post: V3 is not visible to main, foo, and bar.	baz = foo.begin_transaction(); baz.update('doc.txt', '…a=44\n ');
7	git switch foo; git merge baz		V3 becomes visible to foo, but not to main and bar.	baz.commit();
8	git switch main; git merge foo		V1 and V3 become visible to main, but not to bar.	foo.commit();
9	git merge bar		May not work: bar needs to check for conflict with foo.	bar.commit();

However, Git's commit graph, the mechanism that enables branching, offers greater flexibility than database transactions by providing fine-grained control over the snapshot from which a branch can be forked. The developer can fork the repository from any active branch, not just the main branch. To provide this flexibility, GenericVC needs to support Git-like branching where transactions (branches) can be started not only from the public snapshot (main branch) of the data but also from active transactions. We use the concept of nested transactions to support this feature. In doing so, we also utilize the transaction processing mechanisms of a database to manage Git-style (long-running and durable) branches instead of introducing a separate branching layer on top of the database. The nested transaction model [9, 13] was first proposed by J. E. Moss to generalize the flat transaction model and allow for creating nested transactions within a parent transaction. A branch in Git can be represented as a transaction in GenericVC that can be nested if necessary as shown in Table 1. The example uses the following simple schema for a database that stores documents:

$$[R] = \{[\text{document\_name: char, document\_content: text}]\} \quad (1)$$

In Table 1, lines 2 and 3, two feature branches foo and bar are created (forked) from the main branch. In line 6, a nested transaction baz is created from foo, making foo a parent transaction. Given that the example operates on the snapshot isolation level similar to Git (as discussed in Section 2), versions created by baz will be visible to foo when baz commits in line 8. However, these versions will not be visible to main until foo commits in line 9.

There are two major differences between the nested transactions described by Moss and the nested transactions in GenericVC that must support Git-like branching. The first one is the durability of nested transactions, which is not supported in Moss's nested transaction model. The commit operation in Git creates commit objects that are persisted to the disk immediately even if the branch is not yet merged. To provide Git-style durability, GenericVC exposes a configuration parameter that the user can set (similar to setting an isolation level) for the system to persist the local workspace of a transaction even if it does not commit immediately.

The second difference is the hierarchical commit rules for nested transactions. In Moss's model, for a transaction to commit, all its descendants must be resolved (committed or aborted) and all its ancestors must commit. Similarly, if an ancestor transaction aborts,

all its descendants must abort. In contrast, a feature branch in Git can be forked from another feature branch and be merged back to the main branch even if its parent branch is never merged. In Table 1, this would mean that baz can directly commit to main without foo committing. To support this behavior, GenericVC can find a common ancestor (merge base in Git) between any two transactions, identify the versions created by both histories since the ancestor, and apply the difference to the target transaction. Finding a common ancestor can be achieved by keeping a pointer from each transaction's local workspace to its parent transaction's local workspace. We leave the implementation details of these two differences to future work because it is beyond the scope of this vision paper.

5.2 Isolation Levels

In this section, we explain how isolation levels work in GenericVC in the context of nested transactions to support both Git and MVCC use cases. We use the following notation in our execution examples: $w_{foo}[x]$ and $r_{foo}[x]$ to represent write and read operations to a tuple x in transaction foo, and $b_{foo}[baz]$ to represent the creation of a branch baz from foo.

We start with **snapshot isolation**, the default level for Git, and used in the example in Table 1. At this level, a transaction can only read the versions committed before the transaction started. In a nested transactional model, this means that a child transaction can only read the versions that were visible to its parent at the beginning of the child transaction. In line 6 in Table 1, nested transaction baz is created from foo, and it can read the versions visible to foo until that point but not the versions created by or that become visible to foo after that point. To give a more concrete example, if we have the following schedule "$w_{foo}[x]\ b_{foo}[baz]\ w_{foo}[y]$", baz can read the new version of x but not the new version of y since the version was created after baz started. Note that the new version of x is visible to baz even if foo has not committed, because foo is the parent of baz. If instead x was updated by a sibling of baz, the sibling would need to commit (make its versions visible to foo) before baz could read the new version of x.

In **repeatable read**, the transaction can read a version that was committed after the transaction started, but once a specific version has been read, the same version should be read throughout the transaction. In the nested model, this means that a child can read the latest version of a tuple that was visible to its parent until the child transaction reads that tuple for the first time. In the execution "$b_{foo}[baz]\ w_{foo}[y]\ r_{baz}[y]\ w_{foo}[y]$", baz can read the version of y created by the first write operation but not the second.

For the least strict levels of **read uncommitted** and **read committed**, a child transaction can read any version of a tuple that is visible to its parent. The only difference between these two levels is that the meaning of visibility to the parent transaction changes. In read committed, a visible version is a committed version (committed by other children), whereas in read uncommitted, a visible version can be any version, even if it is not committed.

For **serializable isolation**, in addition to setting the isolation level to snapshot isolation (or repeatable read in some DBMSs), we also need to tackle write-write and read-write conflicts which we discuss next.

5.3 Conflict Resolution: Prevention, Detection, and Reconciliation

Conflict resolution is a crucial part of transaction processing in both Git and MVCC and it consists of three main steps: conflict prevention, conflict detection, and conflict reconciliation. Conflict prevention ensures that two transactions are not allowed to write to the same tuple concurrently in the first place. Git does not have a conflict prevention mechanism and handles all conflicts during the merge operation (commit validation phase in databases). MVCC systems often divide conflicts into two categories, write-write and read-write, and prevent write-write conflicts before the commit validation phase via write locks [10, 14, 20]. To accommodate both Git and MVCC use cases, GenericVC exposes a boolean parameter `prevent_ww_conflicts` that can be configured. When set to *false*, this parameter would allow two concurrent transactions to write to the same logical tuple (creating two physical versions) without aborting the second writer, supporting Git-style conflict resolution. When set to *true*, the system would prevent write-write conflicts, via first-writer-wins policy, before the commit validation phase, supporting MVCC-style conflict resolution. We leave the implementation details of this parameter to future work, as it is beyond the scope of this vision paper.

During the commit validation phase, both MVCC and Git detect conflicts between the transaction to be committed and the transactions that have been committed since the start of the transaction. In addition to detecting conflicts, Git also tries to reconcile the conflicting changes automatically. We address both these requirements in GenericVC during the commit validation phase. A high-level implementation of the commit validation function is given in Algorithm 1.

Algorithm 1 Implementation of commit validation phase.

```
1:  function COMMIT_VALIDATION(T, validation_strategy)
2:      // Get the tuple versions committed to T's parent after T started.
3:      parent_write_set = T.parent.get_versions_committed_after(T.begin_ts)
4:      // Check for conflicts between T's read and write sets and parent_write_set.
5:      // Read set is only considered in serializable isolation level.
6:      conflicting_versions = validation_strategy.DETECT_CONFLICTS(
            T.write_set + T.read_set, parent_write_set)
7:      // Reconcile until no more conflicting versions produced.
8:      while conflicting_versions ≠ [] do
9:          // Reconciliation requires further modifications to the data store,
10:         // hence it is regarded as a nested transaction of T.
11:         T_rec = T.start_transaction()
12:         failed_reconciliations = validation_strategy.RECONCILE(
                T, T.parent, conflicting_versions, T_rec)
13:         if T_rec.commit() == FAILURE ∨ failed_reconciliations ≠ [] then
14:             return FAILURE // Abort T.
15:         else // Rerun conflict detection for new (reconciled) versions.
16:             conflicting_versions = validation_strategy.DETECT_CONFLICTS(
                    T_rec.write_set + T_rec.read_set, parent_write_set)
17:     return SUCCESS // Write T's write set to the log and the data store.
```

This commit validation mechanism detects conflicts, reconciles them, and then recursively detects conflicts again on the versions created during the reconciliation process until no more conflicts are detected. The commit validation function takes two arguments: the transaction to be committed (T) and a validation strategy that defines how conflicts are detected and reconciled. For each use case (Git, MVCC, or hybrid), the developer can define a different validation strategy that implements the `detect_conflicts`

and `reconcile` functions. In line 6, `detect_conflicts` function is called to detect conflicts between the versions read and written by T and the versions committed to T's parent transaction (T_parent) since T's start. Reconciliation process can produce new tuple versions, thus it is regarded as a nested transaction of T. In line 12, any new version produced by `reconcile` is committed to this nested transaction. If `reconcile` returns any versions that requires manual handling or if the nested transaction fails to commit for any reason, the transaction T is aborted (lines 13, 14). In line 16, upon successful reconciliation, `detect_conflicts` is executed again on the new versions to ensure that reconciliation does not introduce new conflicts. If any new conflicts are detected, the process is repeated until no more conflicts are detected.

Algorithm 2 Example implementation for DETECT_CONFLICTS.

```
1: function DETECT_CONFLICTS(
       T_read_write_set, T_parent_write_set)
2:     conflicting_versions ← [ ] // Versions requiring reconciliation.
3:     // tid is assumed to be a composite key of table_id and row_id.
4:     for tid ∈ T_read_write_set ∩ T_parent_write_set do
5:         v_T = T_read_write_set[tid] // T's version of the tuple.
6:         v_T_parent = T_parent_write_set[tid] // T_parent's version.
7:         append (v_T, v_T_parent) to conflicting_versions
8:     return conflicting_versions
```

Both functions (`detect_conflicts` and `reconcile`) can be defined as per relation, transaction, or database, depending on the granularity of the conflict detection and reconciliation required by the application. An example implementation of `detect_conflicts` is given in Algorithm 2. In line 4, the implementation detects conflicts at the tuple level by tracking the identifiers (tid) of tuples that were concurrently modified by the committing transaction (T) and the siblings previously committed to T's parent. If a tuple is concurrently modified, the algorithm marks both versions in T and T_parent as "conflicting" (line 7) and returns these versions (line 8) for `reconcile` to handle. For other use cases, the function can be customized to detect conflicts at the attribute level or even finer granularity. We provide an example use case for attribute-level conflict detection in Section 6.

The goal of `reconcile` is to commit new (reconciled) versions to the transaction T and invalidate the versions conflicting with what is previously committed to T's parent. We provide two example implementations of `reconcile` for Git and a hybrid use case in Algorithm 3. The implementation of the reconciliation function for a hybrid use case (`reconcile_inventory_table`) will be discussed in Section 6. The reconciliation function for Git (`reconcile_git`) identifies concurrent delete-delete and insert-update operations made by a committed T_sibling and T as non-conflicts and accepts T's versions (line 4). If there is an update-update conflict, the algorithm tries to reconcile conflicts on the same attribute (`document_content`) using Git-style merge (`merge`) and creates a new reconciled version of the tuple if Git-style merge is successful (lines 6–11). Any other type of conflict is marked as failed reconciliation as Git does not support automatic reconciliation for these cases (line 13).

Incorporating the Git-style conflict reconciliation into GenericVC enables the handling of new hybrid use cases, especially in the context of MVCC workloads. Some of these hybrid use cases

Algorithm 3 Example implementations for RECONCILE for Git and hybrid use cases.

```
1: function RECONCILE_GIT(T, T_parent, conflicting_versions, T_rec)
2:     failed_reconciliations ← [ ] // Versions requiring manual handling.
3:     for v_T, v_T_parent ∈ conflicting_versions do
4:         if (v_T.del ∧ v_T_parent.del) ∨ (v_T_parent.ins ∧ v_T.upd) then
5:             continue // Not considered conflicts for Git.
6:         else if v_T.upd ∧ v_T_parent.upd then
7:             // Get a version before T started, to use as reference for reconciliation.
8:             v_Base = T_parent.get_version_as_of(T.begin_ts, v_T.tid)
9:             // Run Git's 3-way merge.
10:            result, new_version = MERGE(v_T.document_content,
                   v_T_parent.document_content, v_Base.document_content)
11:            if result == SUCCESS then T_rec.insert(new_version) continue
12:        // Git resolves other cases manually.
13:        failed_reconciliations.append((v_T, v_T_parent))
14:    return failed_reconciliations
15:
16: function RECONCILE_INVENTORY_TABLE(T, T_parent, conflicting_versions, T_rec)
17:    // Example strategy for use case in Table 2.
18:    failed_reconciliations ← [ ]
19:    for v_T, v_T_parent ∈ conflicting_versions do
20:        v_Base = T_parent.get_version_as_of(T.begin_ts, v_T.tid)
21:        // v_T: <Elden Ring, 0>; v_Base: <Elden Ring, 1>
22:        // v_T_parent: <Elden Ring, 0> -> public version after Bob's commit.
23:        final_inventory_count = v_T_parent.count - (v_Base.count - v_T.count)
24:        if final_inventory_count < 0 then // Reconcile based on stock.
25:            version_SC = T_rec.select(... get tuple from Shopping Cart Table ...)
26:            T_rec.delete(version_SC) // Remove from Shopping Cart.
27:            T_rec.insert(... new version in Wishlist Table ...)
28:            T_rec.delete(v_T) // Delete, as not valid after reconciliation.
29:        else
30:            // Developer can handle more conflict types here if needed.
31:            continue
32:    return failed_reconciliations
```

are identified by prior work [3, 4, 17], where instead of blindly aborting a transaction in the event of a conflict, the application layer tries to reconcile conflicting changes. These studies aim to resolve these conflicts at the application layer, however, we argue that conflict resolution should be handled at the database layer to maintain the separation of concerns principle. Our goal is to handle most of these conflicts as part of the commit validation phase of a transaction. In the best case, `reconcile` can be defined per relation and release the application layer from the burden of conflict resolution. In the worst case, `reconcile` should sometimes be defined per transaction for more specific use cases in the application layer. This would require more fine-grained customization for conflict resolution, but the database layer would still do most of the heavy lifting (reconciliation) instead of the application sending multiple queries to the database back and forth for reconciliation.

We provide an example implementation (`reconcile_inventory_table`) for a per-relation use case and elaborate on the example in the upcoming Section 6. Per-transaction reconciliation requires developers to define reconciliation logic alongside the transaction body. While this concept resembles database triggers, the existing trigger implementations cannot support reconciliation within the commit validation phase. Some DBMSs such as PostgreSQL and Oracle support deferred triggers, however, the trigger function is still executed at the end of the transaction before the validation phase starts. If reconciliation logic was executed via a deferred trigger, another concurrent transaction could execute validation and commit earlier, invalidating the reconciliation phase of the current transaction. Thus, the reconciliation logic should be executed within

the commit validation phase to ensure that no other concurrent transaction can commit while the reconciliation is in progress.

6 BEYOND A RELATIONAL DATABASE WITH BRANCHING CAPABILITIES

In this section, we present two hybrid use cases that require features from both Git and MVCC.

Use-Case 1: Imagine an online store that sells computer games. In a busy day, Bob and Alice want to buy the same game and they both see a warning that only one copy is left in stock. They add the game to their respective shopping carts. Alice additionally adds another game. The current state of the online store is shown in Table 2 in black text. Now, Bob and Alice both click the purchase button. Both transactions, T_{Bob} and T_{Alice} start back-to-back, and they both read the same state of the Game Inventory Table. T_{Bob} commits first, so now the game Elden Ring is out of stock, and T_{Alice} needs to be dealt with. A traditional database would abort T_{Alice} due to a write-write conflict, retry the transaction, abort again due to out-of-stock check constraint, and finally give up, leaving the items in Alice's cart. To provide a better user experience, most online stores implement application layer logic such that when T_{Alice} aborts due to the out-of-stock game, another request to database is sent to move the out-of-stock game to a wish list for future purchase, process the rest of the items in the cart, and notify Alice that the particular game is out of stock. This approach results in two transaction aborts and an extra round trip to the application layer for reconciliation.

GenericVC can do *all of this with no aborts and no extra round trip to the application layer by implementing a custom reconciliation strategy* for the Game Inventory Table as shown in the example implementation in Algorithm 3 (`reconcile_inventory_table`). This reconciliation strategy assumes that `prevent_ww_conflicts` is set to *false* in GenericVC configuration, thus two concurrent transactions can simultaneously update the same tuple and any conflicts will be resolved during commit validation phase. In line 23, the tuple in the Game Inventory Table of Elden Ring has three versions: *v_T* belongs to T_{Alice}, *v_T_Parent* belongs to T_{Bob}, and *v_Base* is the version that existed before both transactions started. In line 24, the strategy checks if the stock count of the game drops below zero if T_{Alice} is committed, and moves the existing orders to the Wishlist Table (lines 26–28) if it does. The resulting state of the online store after both checkouts is shown in Table 2 in red text. Therefore, the reconciliation strategy in GenericVC can handle the out-of-stock scenario without aborting any transactions and without any round trips to the application layer.

Use-Case 2: Assume a large dataset where multiple data analysts collaborate to clean and prepare data to run machine learning algorithms. Analyst Bob updates a column in a table to use a different unit of measurement while analyst Alice updates another column to use a different scale and deletes some tuples with missing values. If T_{Bob} committed first, T_{Alice} would be aborted and re-tried in MVCC. However, in the context of collaborative data cleaning, Alice's changes are not necessarily in conflict with Bob's changes, and accepting them in the first place could help decrease abort rates, especially in high contention scenarios. In GenericVC, `detect_conflicts` can be defined to handle attribute-level reconciliation by comparing attribute ids instead of tuple ids, so that changes to different columns are not considered in conflict. Additionally, `detect_conflicts` can prefer deletions over updates, so that if a tuple is updated in one transaction and deleted in another subsequent transaction, the deletion in the latter transaction is still committed. While attribute-level conflict detection can be implemented in existing MVCC and locking-based systems, preference for deletions over updates is a semantic conflict resolution strategy that requires context-aware conflict resolution. GenericVC enables such context-aware conflict resolution strategies to be implemented in the database layer either per table or per transaction for specific use cases.

Table 2: Game store before Bob and Alice checkout (in black). Red text shows reconciled state after both checkouts.

Game Inventory Table

game	count
Elden Ring	~~1~~ 0
Cyberpunk 2077	~~5~~ 4

Shopping Cart Table

user	game	count	
Bob	Elden Ring	1	removed
Alice	Elden Ring	1	removed
Alice	Cyberpunk 2077	1	removed

Wishlist Table

user	game	count
Alice	Elden Ring	1

7 SUMMARY AND FUTURE WORK

In this paper, we conducted a comparative analysis of Git and MVCC, and have concluded that the features of both systems can be combined into a single system. We proposed GenericVC for this purpose, a flexible database and a version control system that can serve a range of use cases from Git to MVCC to hybrid ones. The key advantage of GenericVC is that it can handle a wide range of use cases without the need for extra branching functionality or additional application layer logic to handle Git-style branching and MVCC-style concurrency control. We also question the exiting conflict handling of MVCC which should be tunable similar to isolation levels. For that we introduce two user-definable functions that can be run by the DBMS at the validation phase to determine how a conflict is defined and how it should be reconciled.

GenericVC is already under development, and we plan to conduct experiments to evaluate its performance and scalability in future work. Allowing extensive configuration in GenericVC, especially during conflict resolution, opens up a new research direction as user-defined conflict resolution strategies would affect everything from query performance to consistency guarantees and the system's correctness. Another research direction to explore is if the concurrency control mechanisms of GenericVC can be fully decoupled from the storage layer and be used as a concurrency-control-as-a-service for any storage system. Abstracting concurrency control mechanisms from the storage layer can simplify development and improve its flexibility and scalability. Since GenericVC is a highly configurable system, we also plan to investigate the ways to automatically tune it to provide the optimal performance for a given workload. There is also extensive research on automatic tuning of

databases [1, 6, 11], but most of these studies focus on low-level parameters such as memory and cache distribution, query planning, and logging. In GenericVC, we also would like to explore how to automatically tune the high-level parameters such as nested transaction behavior, isolation levels, conflict resolution and locking strategies, that define the system's architecture and behavior.

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD Proceedings*. ACM, 1009–1024.

[2] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD Proceedings*. ACM, 729–738.

[3] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *SIGMOD Proceedings*. ACM, 1615–1628.

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *SOSP Proceedings*. ACM, 205–220.

[5] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Proceedings*. ACM, 1243–1254.

[6] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.

[7] Git. 2005. https://git-scm.com/.

[8] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya G. Parameswaran. 2017. OrpheusDB: Bolt-on Versioning for Relational Databases. *Proc. VLDB Endow.* 10, 10 (2017), 1130–1141.

[9] George Karabatis. 2009. Nested Transaction Models. In *Encyclopedia of Database Systems*. Springer US, 1896–1899.

[10] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (2011), 298–309.

[11] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.

[12] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya G. Parameswaran, and Amol Deshpande. 2016. Decibel: The Relational Dataset Branching System. *Proc. VLDB Endow.* 9, 9 (2016), 624–635.

[13] J Eliot B Moss. 1985. *Nested transactions: An approach to reliable distributed computing*. Massachusetts Institute of Technology.

[14] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD Proceedings*. ACM, 677–689.

[15] PostgreSQL. 1986. https://www.postgresql.org/.

[16] Nalin Ranjan, Zechao Shang, Sanjay Krishnan, and Aaron J. Elmore. 2021. Version Reconciliation for Collaborative Databases. In *SoCC Rec.* ACM, 473–488.

[17] Maximilian E. Schüle, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. 2019. Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In *SSDBM Proceedings*. ACM, 169–180.

[18] Maximilian E. Schüle, Josef Schmeißer, Thomas Blum, Alfons Kemper, and Thomas Neumann. 2021. TardisDB: Extending SQL to Support Versioning. In *SIGMOD Rec.* ACM, 2775–2778.

[19] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.* 26, 4 (2017), 537–562.

[20] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.